Aspects du système APL 90 : Une extension orientée objet du langage APL

Séga Sako

To cite this version:

Séga Sako. Aspects du système APL 90 : Une extension orientée objet du langage APL. Intelligence artificielle [cs.AI]. Ecole Nationale Supérieure des Mines de Saint-Etienne; Institut National Polytechnique de Grenoble - INPG, 1986. Français. NNT : . tel-00829941

HAL Id: tel-00829941

https://theses.hal.science/tel-00829941

Submitted on 4 Jun 2013

ECOLE NATIONALE SUPERIEURE
DES MINES DE SAINT-ETIENNE

UNIVERSITE DE
SAINT-ETIENNE

N° d'ordre : 5 ID

THESE de DOCTORAT

présentée par

Séga SAKO

Spécialité :

Informatique, Image, Intelligence Artificielle et Algorithmique

ASPECTS DU SYSTEME APL 90

UNE EXTENSION ORIENTEE OBJET DU LANGAGE APL

Soutenue à Saint-Etienne le 5 Décembre 1986, devant la commission d'examen :

Président	M. HABIB
Examinateurs	J.J. GIRARDOT
	J.P. GOURE
	M. LENCI
	G. MARTIN
	F. MIREAUX
	B. PEROCHE

Thèse préparée au sein du Département Informatique de l'Ecole des Mines de Saint-Etienne.

ECOLE NATIONALE SUPERIEURE
DES MINES DE SAINT-ETIENNE

UNIVERSITE DE
SAINT-ETIENNE

N° d'ordre : 5 ID

THESE de DOCTORAT

présentée par

Séga SAKO

Spécialité :

Informatique, Image, Intelligence Artificielle et Algorithmique

ASPECTS DU SYSTEME APL 90

UNE EXTENSION ORIENTEE OBJET DU LANGAGE APL

Soutenue à Saint-Etienne le 5 Décembre 1986, devant la commission d'examen :

Président	M. HABIB
Examinateurs	J.J. GIRARDOT
	J.P. GOURE
	M. LENCI
	G. MARTIN
	F. MIREAUX
	B. PEROCHE

Thèse préparée au sein du Département Informatique de l'Ecole des Mines de Saint-Etienne.

REMERCIEMENTS

Qu'il me soit permis d'exprimer ici mes plus vifs remerciements, à tous ceux qui ont accepté de juger ce travail ainsi qu'à ceux qui m'ont aidé à le mener à bien.

A Monsieur le Professeur Michel HABIB de l'Ecole Nationale Supérieure des Télécommunications de Bretagne, qui me fait l'honneur de présider le jury de cette thèse. Il a su communiquer au Département Informatique alors qu'il en était le directeur, son enthousiasme et son dynamisme pour le développement des équipes de recherche. Manifestant beaucoup d'intérêt pour les L.O.O (Langages Orientés Objets), il a accepté de juger ce travail.

A Messieurs Jean Jacques GIRARDOT et François MIREAUX que je ne saurai dissocier. C'est à eux que je dois d'avoir vécu cette grande aventure. Sans l'inspiration du premier, cette thèse serait sans objet, et sans la rigueur du second APL 90 aurait eu quelques trous de mémoire (virtuelle bien sûr) ! Ils m'ont accueilli au sein de leur équipe de recherche, et m'ont permis de participer au projet APL 90 qu'ils ont élaboré et conduit en commun. Sans leur grande expérience, leur compétence et leur infinie patience, un tel travail n'aurait pu aboutir. Ils ont suivi cette thèse de sa génèse à son terme et j'ai toujours trouvé auprès d'eux des conseils scientifiques éclairés, des critiques constructives et des encouragements sans mesure.

A Monsieur le Professeur Jean Pierre GOURE, Directeur du Laboratoire TSI (UA 842) de l'Université de Saint-Etienne pour sa participation à ce jury.

A Monsieur le Professeur Michel LENCI, Directeur du Centre d'Automatique et d'Informatique de l'Ecole des Mines de Paris, pour la lecture de ce travail et pour sa présence aujourd'hui.

A Monsieur le Professeur Bernard PEROCHE, Directeur du Département Informatique de l'Ecole des Mines de Saint-Etienne, pour son aide et pour l'intérêt qu'il a porté à cette thèse.

A Monsieur Gilles MARTIN, Président du Collège Informatique de l'AFCET, animateur du groupe de travail APL du même organisme, pour avoir accepté d'être membre du jury et apporter ainsi toute sa compétence dans le domaine que nous traitons.

A mes ainés (Christian, Ulysse ...) et ma cadette (Florence) de l'équipe *Langages Interactifs* pour l'héritage qu'ils nous ont légué dans le domaine du langage APL.

A tous les membres présents et passés du Département Informatique de l'Ecole des Mines de Saint-Etienne pour la bonne ambiance, leur amitié, leur aide durant ces années de thèse.

Aux promotions d'élèves de la SSEI et de l'Ecole des Mines qui ont subi avec abnégation l'utilisation du système APL 90 comme support de cours et de TP, et qui ont ainsi facilité la mise au point du logiciel.

A mes collègues du groupe de travail APL de l'AFNOR pour toutes les discutions fructueuses autour du standard-ISO du langage APL.

A Marie Line BARNEOUD qui n'a jamais ménagé ses efforts et son temps pour m'aider à saisir ce document. Sa force de frappe a été très dissuasive.

A Messieurs DARLES, LOUBET et VELAY du service de reprographie de l'Ecole des Mines qui ont toujours fait le maximum pour la réalisation de cet ouvrage.

A GROFF[1], UNIX[2], SM 90[3] (qui se porte très bien) et LaserWriter[4] pour avoir constitué l'environnement matériel et logiciel de cette thèse.

Et enfin à ma famille et à mes amis, pour leur patience, leur soutien et leur réconfort. C'est peu que de leur dédier ce travail.

---

[1] **groff** est un formatteur de texte disponible sous UNIX et distibué par la société **sync** à Saint-Etienne.

[2] UNIX est une marque déposée des laboratoires A.T & T.

[3] SM 90 est une architecture de machine conçue par le CNET et disponible chez les constructeurs sous divers noms (SPS7, SM 90 TELMAT, ...)

[4] LaserWriter est une imprimante à Laser de la société APPLE.

SOMMAIRE

.... PREMIERE PARTIE ....

CHAPITRE 1

CHAPITRE 2

.... DEUXIEME PARTIE ....

CHAPITRE 3

CHAPITRE 4

CHAPITRE 5

CHAPITRE 6

.... CONCLUSION ....

.... BIBLIOGRAPHIE ....

.... ANNEXES ....

INTRODUCTION

1 AVANT PROPOS

Le langage APL a été défini en 1962 par K.E. Iverson dans un ouvrage intitulé " A Programming Language " [Iverson62]. En dépit des difficultés nouvelles que ce langage posait aux concepteurs, des interprètes APL furent rapidement disponibles chez I.B.M. [Chomat71, Iverson73].

Ce langage s'étant d'emblée révélé intéressant pour toutes les applications réclamant plus de programmation que de longs calculs, plusieurs systèmes virent le jour chez d'autres constructeurs [Martin72, Girardot76]. Réalisés pour la plupart sur de gros ordinateurs, ils réclament des ressources hors de proportion avec les moyens habituels mis à la disposition des ingénieurs ou des étudiants.

Parallèlement de nombreux travaux de recherche ont été menés afin d'accélérer l'interprétation du langage. P.S. Abrams fut le premier à introduire certaines notions fondamentales d'optimisation comme le *beating* devenu classique de nos jours. Il a également envisagé la construction d'un matériel muni d'un code machine très voisin du langage APL [Abrams70].

D'autres approches ont consisté à réaliser un maximum de fonctions au niveau matériel, en microprogrammant un sous-interprète scalaire ou vectoriel, et en rédigeant ensuite, dans l'APL restreint ainsi obtenu, l'interprète complet.

L'étude des divers systèmes existants montre que les architectures purement logicielles, utilisant presque toujours une technique d'interprétation naïve sont les plus répandues. Cependant, même si les architectures sont restées identiques, l'évolution des techniques d'implantation a permis des progrès considérables.

De fait, il semble assez difficile de progresser beaucoup dans une voie purement logicielle si ce n'est par des approches radicalement différentes comme celles décrites dans [Hewlett77]. Ce système APL réalisé sur HP 3000 est en fait un compilateur incrémental, générant pour chaque ligne de code qu'il exécute pour la première fois, un fragment de code machine. Une signature associée à ce code permet de savoir ultérieurement si la réutilisation de ce code est possible.

Par ailleurs, depuis quelques années diverses, extensions du langage APL ont vu le jour : les unes importantes affectant la totalité du langage ou du système, les autres ponctuelles visant quelques aspects spécifiques. On peut citer notamment :

- l'introduction de la notion de tableaux arborescents et de fonctions spécifiques de traitement de ces tableaux,
- l'extension de la portée des fonctions primitives,
- la possibilité de définition d'opérateurs par l'utilisateur,
- la gestion des interruptions et des évenements.

Il existe à l'heure actuelle deux implantations majeures : APL2 d'IBM [Brown86] et SHARP-APL [Berry79, Iverson83], qui proposent des extensions importantes et parfois contradictoires.

Malgré ces extensions, les réalisations existantes souffrent de limitations importantes. Citons en quelques unes :

- certains concepts du langage, comme la notion de zone de travail, sont restés trop limitatifs par rapport aux besoins sans cesse grandissants des utilisateurs. On ne trouve pas dans APL d'environnement de programmation comparable à ceux qui existent autour de SMALLTALK ou des machines LISP,
- le fait de passer par un système d'exploitation "général" souvent mal adapté aux besoins d'APL, entraîne la dégradation des performances des systèmes APL,
- les possibilités, au niveau de l'architecture des machines, offertes par la venue de microprocesseurs à la fois souples, puissants et peu onéreux, ne sont pas exploitées.

C'est dans ce contexte que l'équipe "langages interactifs" du Département Informatique de l'Ecole Supérieure des Mines de Saint-Etienne a décidé d'élaborer le projet APL 90.

Débuté en 1982, ce projet a pour but de permettre une réflexion sur le langage et le système APL à travers plusieurs phases. Alors qu'un groupe de travail se constituait pour l'élaboration d'une norme au sein de l'ISO, groupe dans lequel s'intégraient les membres de l'équipe, et qui avait pour tâche de déterminer exactement ce qu'était le langage APL, il paraissait intéressant de réfléchir également à ce qu'il pourrait devenir.

Des réflexions ont ainsi été menées dans plusieurs directions :

- l'architecture logicielle d'un système APL,
- l'architecture matérielle d'une machine APL,
- la syntaxe du langage,
- la recherche du parallélisme dans le langage,
- les extensions proposées à l'heure actuelle essentiellement par APL2 de J. Brown et RATIONALISED-APL de K. Iverson,
- les extensions qui sont à proposer en s'inspirant notamment des langages orientés objet.

Tous ces aspects ne seront pas abordés dans le travail que nous présentons ici. Notre but correspondant à la première phase du projet couvrira les points suivants :

- étude d'une architecture logicielle de machine que nous appelerons ALM 90 permettant la réalisation aisée de langages de traitement de tableaux. C'est le langage APL qui nous intéresse en premier, mais d'autres langages de la même famille pourront être envisagés,
- réalisation sur une petite machine d'un système APL conforme à la norme ISO. Ce système bâti suivant le modèle de l'ALM 90 doit être :
    - complet, fiable, sans réelles limitations
    - performant
    - extensible
- l'implantation d'une extension orientée objet.

Ce système que nous appellerons dans la suite APL 90, a été effectivement réalisé sur un ordinateur SM 90 à l'Ecole Supérieure des Mines de Saint-Etienne. Outre la programmation du système APL 90 qui a été principalement réalisée en collaboration avec Jean-Jacques Girardot et François Mireaux, les travaux de l'auteur de cette thèse ont porté plus particulièrement sur les points suivants :

- Elaboration d'une gestion de mémoire virtuelle pour APL 90.
- Intégration des fonctions primitives du langage.
- Etude d'une extension orientée objet du langage et du système APL.

- Définition d'une norme ISO pour APL et élaboration d'une version française dans le cadre du groupe de travail AFNOR-APL [Norme86].

La description complète d'un tel travail ne rentre pas dans le cadre d'une thèse et ne serait que d'un interêt relatif. La thèse ne contiendra que certains des aspects les plus intéressants de la contribution de l'auteur, aspects qui, de ce fait, ont pu être exposés plus précisément.

2 FINANCEMENT DES RECHERCHES

Les organismes suivants ont participé financièrement au projet :

- Le Ministère de la Recherche et de l'Industrie pour une partie du matériel et les frais de personnel.
- L'Agence De l'Informatique, pour l'octroi d'une machine SM 90 bien configurée et une partie des frais de fonctionnement.

3 PLAN DE LA THESE

Cette thèse comporte deux parties.

La première partie comprend deux chapitres qui sont consacrés spécifiquement à certains aspects du système APL 90.

Le chapitre 1 commence par une introduction brève au langage et au système APL. Les concepts de base du langage, la syntaxe ainsi que l'environnement d'exécution y sont exposés. Il se poursuit par une étude détaillée des systèmes APL existants, étude qui montre les limites de ces systèmes. Ce chapitre se termine par l'approche d'APL 90 et la définition d'une architecture logicielle de machine (ALM 90).

Le deuxième chapitre traite des extensions qui sont en "vogue" dans la communauté APL, et que nous qualifierons d'extensions "classiques". Il débute par un bref survol de certaines propositions d'extension. Ensuite nous définissons quelques critères qui nous semblent importants pour l'intégration des extensions. Certaines de ces extensions étant contradictoires, nous avons analysé leur essence. Cette analyse a porté sur trois classes d'extension : les tableaux généralisés, la représentation vectorielle ou *strand notation,* et les opérateurs. Nous étudierons tout d'abord les extensions de type APL2 puis les opérateurs d'Iverson. L'intégration des extensions est discutée.

La seconde partie recouvre l'extension orientée objet. Les paradigmes de programmation que l'on rencontre de nos jours sont classés en trois groupes : la programmation orientée procédure, règle et objet. Le chapitre 3 décrit les concepts de base des langages de programmation orientés objet (que nous désignerons par L.O.O) tels que : objet, instance, classe, transmission de message, héritage. Dans le chapitre 4, un modèle d'APL orienté objet (OBJAPL 90) est proposé selon le modèle de Smalltalk. L'étude détaillée des postulats de ce modèle permet de faire l'inventaire des problèmes liés à l'intégration des concepts de la programmation orientée objet (P.O.O) dans un environnement APL. Des solutions sont proposées au fur et à mesure que ces difficultés apparaissent.

Le chapitre 5 expose une application modeste (l'arithmétique des entiers en précision infinie) qui illustre les mécanismes de la programmation orientée objet dans APL. Les modifications ou les contraintes que la compatibilité avec le système APL2 impose à l'extension objet dans notre système OBJAPL 90 sont explicitées.

Le dernier chapitre est consacré aux structures de données utiles à notre réalisation. Cette partie se termine par la critique de notre modèle et une tentative de réflexion sur l'avenir de la programmation orientée objet en APL.

introduction

PREMIERE PARTIE : ASPECTS DU SYSTEME APL 90

Nous présenterons brièvement dans le chapitre 1 le langage et le système APL. Et pour mieux saisir l'approche d'APL 90, nous étudierons les systèmes existants au niveau de leur architecture. Ce chapitre se termine par la définition de l'architecture logicielle qui supporte le système APL 90 (ALM 90).

Au fur et à mesure que le langage se répandait, certaines de ses faiblesses ont été mises à nu :

- ses faibles performances comparées à celle des langages compilés
- les formes anormales ou irrégulières de la définition originelle d'APL
- le besoin d'extension à des domaines nouveaux.
- la nécessité d'uniformiser certains concepts du langage.

En conséquence, plusieurs propositions d'extension virent le jour. Dans certains cas, ces propositions reflètent les besoins propres de leurs auteurs ou d'un domaine d'application particulier. Dans d'autres cas elles constituent un apport essentiel dont l'adoption enrichit le langage mais en même temps influence la syntaxe ou la nature des objets du langage.

Certaines de ces extensions ont vu le jour dans les nouvelles implantations de système APL, tandis que d'autres sont toujours matière à discussion dans la communauté APL.

C'est pourquoi dans le chapitre 2, nous établirons des critères pour l'extension, et analyserons les aspects intéressants des extensions relatives à trois entités du langage : les tableaux, les fonctions et les opérateurs. Ce sont des extensions que nous qualifierons de "classiques" pour marquer la distinction avec la seconde partie qui traite de l'extension orientée objets. Nous étudierons d'abord les extensions de type APL2, puis les opérateurs d'Iverson et enfin nous discuterons de leur intégration dans le système APL 90.

introduction partie 1

CHAPITRE 1

Chapitre 1

LES AUTRES SYSTEMES ET APL 90

1 PRESENTATION BREVE DU LANGAGE ET SYSTEME APL

1.1 LE LANGAGE APL

Le langage APL est une notation mathématique abstraite née des travaux de K.E. Iverson et A.D. Falkoff sur la formalisation des algorithmes [Iverson62, Iverson71, Falkoff73a]. Il est conçu pour la manipulation globale de données structurées en tableaux rectangulaires, denses et homogènes en type. La récursivité, le dynamisme complet des données, dû essentiellement à l'absence de déclaration, et un nombre important de fonctions primitives alliées à un mécanisme simple d'opérateurs (décrivant des opérations générales sur les données), en font un langage de programmation extrêmement puissant.

Des concepts très simples en rendent l'approche aisée. Le langage APL résume un effort accompli en vue de formaliser un certain nombre de notations mathématiques, la notion de fonction primitive recouvrant le sens plus général de fonction mathématique. Iverson justifie lui même son approche dans [Iverson71].

1.2 LA SYNTAXE DU LANGAGE

Nous ne donnerons pas une description complète et détaillée de la syntaxe formelle du langage APL. Il existe des documents très précis définissant un standard [Falkoff79, Norme86], ou traitant de la syntaxe du langage [Rollin83]. Nous rappellerons uniquement les principes syntaxiques de base qui permettent une utilisation immédiate du langage.

Toutes les instructions APL s'évaluent de la droite vers la gauche **sans aucune priorité.** En notation B.N.F (Backus Naur Form), on écrira la syntaxe d'une expression :

```
<expression> :: = <opérande>

                 |<fonction-monadique><expression>

                 |<opérande><fonction-dyadique><expression>
```

|<fonction-niladique>

Un opérande pouvant être à son tour :

<opérande> :: = <constante>

|<variable>

|(<expression>)

Cette définition volontairement simplifiée ne tient compte ni des expressions indicées ni de la syntaxe particulière des opérateurs. Elle introduit cependant la notion de **valence**, une primitive, ou une fonction définie pouvant avoir un ou deux argument(s).

Ainsi, seul le contexte permet de distinguer l'exponentielle :

```
      *1
2.781828
```

de l'élévation à une puissance :

```
      2 * 3
8
```

1.3 LE SYSTEME APL

Beaucoup plus qu'un langage de haut niveau, APL apparaît comme un outil logiciel d'une rare puissance car il s'insère dans un système complet et cohérent.

A l'opposé des traducteurs FORTRAN et COBOL qui reposent sur le système hôte pour de nombreux services, les implantations d'APL fournissent non seulement le support d'exécution de la notation d'APL, mais aussi un éditeur de fonctions, un système de mise au point, une gestion de bibliothèque de programmes, l'accès à des fichers extérieurs, etc. L'ensemble constitue un système autonome que l'utilisateur voit comme une machine APL.

Les premières réalisations d'interprètes sur IBM/1130, puis sur IBM/360 furent l'occasion pour leurs auteurs de concevoir de véritables machines APL [Falkoff73a, Falkoff73b, Falkoff73c]. Les buts poursuivis étaient nombreux :

- rendre cette machine la plus accessible possible en éliminant tout ce qui décourage le non informaticien de la programmation (langage de commande ésothérique, nombreuses étapes intermédaires, difficultés de communication des programmes avec le monde extérieur).

- obtenir une indépendance du calculateur hôte, sur lequel est réalisé cette machine, la plus grande possible. Ce souci inclut la transparence à l'utilisateur des contraintes matérielles de la réalisation (taille de la mémoire centrale, longueur des mots utilisés, représentations numériques, codes des caractères...)

- intégrer harmonieusement au langage tous les services satellites de la programmation que l'utilisateur est en droit d'attendre : gestion des programmes (édition, mise au point, archivage), contrôle complet de l'environnement d'exécution des programmes.

- laisser le plus de travail automatique et de décisions très ponctuelles à la charge de cette machine et non pas à celle de l'utilisateur.

Pour ce dernier, cette recherche du confort réduit à l'essentiel l'investissement intellectuel parasite que nécéssite en général la programmation d'un problème simple.

Le contexte de travail s'appelle zone de travail active. Vue de l'utilisateur cette zone représente un "cahier de brouillon" dans lequel il conserve ses variables, construit et met au point ses fonctions. Quand l'ensemble lui paraît satisfaisant, il peut sauvegarder cet espace actif sous une forme que l'on appelle zone de travail inactive.

Une zone de travail contient aux yeux de l'utilisateur :

- la table des symboles contenant tous les noms qu'il crée

- les valeurs des variables et des fonctions qu'il a défini

- le vecteur d'états (encore appelé pile des contextes) qui est constitué par l'empilement des blocs dynamiques d'exécution.

De plus cette zone de travail est sous le contrôle d'un certain nombre de paramètres servant d'interfaces entre le système APL et la zone active. Ces paramètres correspondent à des objets communs au système et à l'utilisateur.

2 UNE ETUDE DES SYSTEMES EXISTANTS

2.1 LES ARCHITECTURES TRADITIONNELLES

Par architecture de machine APL, nous entendons l'ensemble des mécanismes tant matériels que logiciels, permettant la réalisation effective d'un système APL.

Nous nous intéressons tout d'abord aux machines APL réalisées à partir de matériel standard - ordinateurs classiques - et dont seul le logiciel est spécialisé. Sur ces calculateurs, un système APL est tout simplement un programme, écrit dans un langage quelconque, effectuant des calculs et des entrées-sorties sur un ou plusieurs terminaux.

La figure 1 montre un exemple de réalisation - que nous qualifierons de traditionnelle - sur un calculateur IBM 360 [Falkoff68]. Le système est découpé en deux modules, de fonctionnalités bien définies : le superviseur et l'interprète.

Le superviseur effectue les entrées/sorties sur les différents terminaux connectés au système. Il s'occupe également de l'allocation de l'unité centrale aux divers utilisateurs, et gère les transferts de zones de travail entre le disque de pagination et la mémoire centrale. L'interprète APL réalise lui même les opérations demandées par l'utilisateur.

Lorsque la gestion de mémoire et du temps d'unité centrale est assurée directement par le système d'exploitation, comme dans le système VSAPL (ou APL 2 - 1982 [Dave82]), on aboutit à la réalisation décrite par la figure 2. Ici, les fonctions de base et l'interprétation sont assurées respectivement par deux programmes (CMS et l'interprète proprement dit) qui sont partagés entre l'ensemble des utilisateurs.

La figure 3 décrit une autre réalisation sur petit calculateur (T1600 de la télémécanique, 1974), [Girardot-Mireaux76].

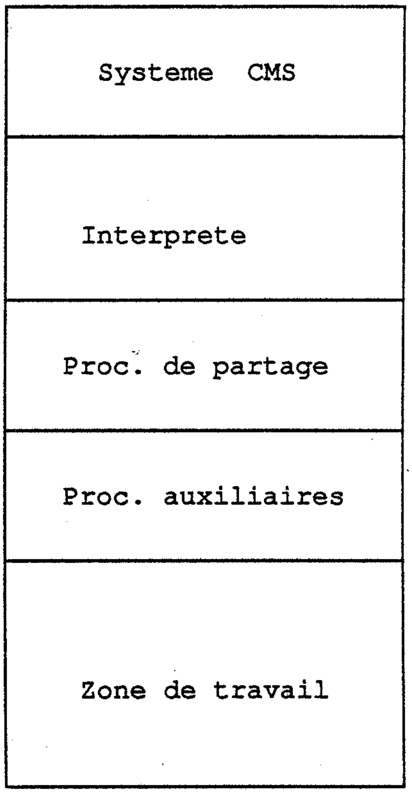
Malgré une taille mémoire réduite (32 k octets) et une faible puissance de calcul, ce système peut gérer une dizaine d'utilisateurs avec de bons temps de réponses, grâce à un système de segmentation simple et l'utilisation d'un disque à tête fixe à temps d'accès très faible.

L'étude des divers systèmes existants montre que les architectures purement logicielles de système APL sont les plus répandues dans la nature. Ceci se justifie par divers impératifs pratiques dont le plus important est que APL ne représente souvent qu'une faible fraction de l'utilisation du calculateur, celui-ci servant principalement à des applications plus traditionnelles.

Cependant, si les architectures sont souvent restées identiques l'évolution des techniques d'implantation a permis des progrès considérables : ainsi, un système comme VSAPL est probablement 2 à 3 fois plus performant que la première implantation réalisée sur le même ordinateur. Ceci s'explique moins par les progrès de la science informatique en général que par une programmation plus fine du système, due à une meilleure connaissance des phénomenes liés à APL, et en particulier à la reconnaissance systématique de cas spécifiques permettant d'accélérer le traitement individuel de presque toutes les fonctions du système. Ainsi la même fonction de recherche d'indice (iota diadique) va être réalisée par dix algorithmes différents, que l'on sélectionnera en fonction de l'aspect des opérandes.

Systeme APL–360 (1968)
256 k octets

Figure 1



Systeme APL 2 (1982)
1 M octets

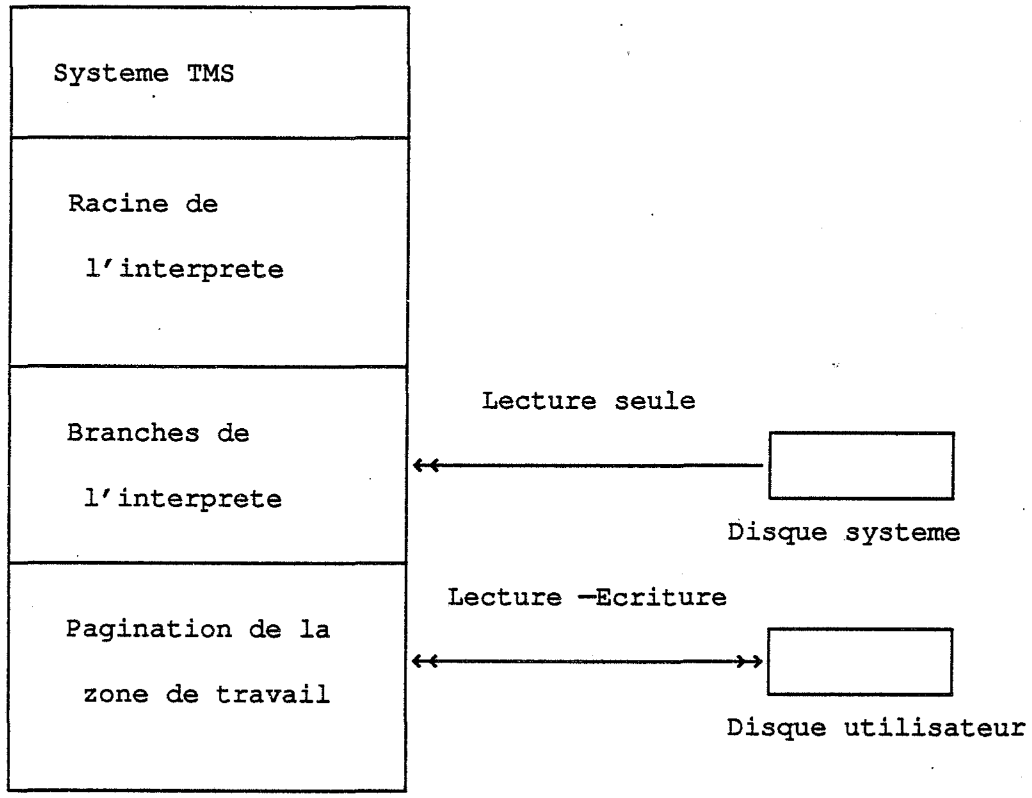
Figure 2

2.2 LES ARCHITECTURES MICROPROGRAMMEES

Devant l'inefficacité - toute relative, il est vrai - des méthodes classiques d'interprétation du langage, grande était la tentation d'améliorer la situation en donnant un petit coup de pouce au matériel.

Les premières recherches dans le domaine, effectuées au début des années 70, s'attaquaient à la seule couche existant entre le matériel et le logiciel : la microprogrammation.

L'une des réalisations typiques de cette approche est celle décrite dans [Hassit73]. Utilisant un calculateur de type IBM 360-25, les auteurs ont cherché à remplacer le micro-programme réalisant le code d'ordre de la série 360 par un autre ensemble de fonctions, mieux adaptées à la réalisation d'APL sur cet ordinateur.



Systeme APL — 1600 (1974) 32 K octets

Figure 3

Une évolution ultérieure [Hassit76] permit la réalisation, sur la série 370-145, d'un microcode spécialisé, "APL assist", prenant entièrement en charge l'interprétation d'APL dans la majorité des cas simples, les autres étant traités par des programmes écrits en assembleur 370 ou en APL (fonctions "magiques").

L'intérêt de cette approche ne semble pas contestable : les tests prouvent suivant les applications considérées, des gains variant de 2 à 5 sur l'interprétation sans utilisation de microcode. Les rares cas qui ne montrent pas d'augmentation sensible de performance sont ceux qui utilisent les nombres flottants ou des fonctions non réalisées par le microcode.

Cependant, l'avenir d'une telle approche nous paraît à terme limité. Remarquons tout d'abord que si le code d'ordre est le même sur tous les calculateurs de la série 370, le microcode d'ordre est spécifique à chaque modèle : fournir "APL assist" pour toute la gamme necessiterait probablement l'écriture de 5 ou 6 versions différentes de ce micro logiciel, en utilisant à chaque fois un microcode différent. En outre porter un microcode d'un calculateur a un autre est moins aisé que de transporter un logiciel : la microprogrammation est beaucoup moins simple que la programmmation, et nécessite une connaissance parfaite de la micromachine et de ses contraintes temporelles. Il s'agirait donc d'une étude presqu'entièrement originale pour chaque nouveau modèle envisagé. C'est là un investissement qui mérite réflexion. Enfin une dernière considération est à prendre en compte : si, dans la série 370, les modèles de bas de gamme sont effectivement microprogrammés, ceux du haut de gamme réalisent directement le code d'ordre par du silicium... Il n'est dès lors plus question de microprogrammation...

2.3 LES MACHINES APL

On peut à ce stade dire quelques mots des "machines APL" actuellement disponibles dans le commerce, telle la série des IBM 5100, ou MCM 700. Ces calculateurs offrent exclusivement le langage APL (plus ou moins complet selon les systèmes). Ce sont bien des machines APL dans le sens qui nous intéresse. Cependant leur architecture est similaire à celles que nous avons déjà étudiées ; le système APL est réalisé soit entièrement en logiciel, soit partiellement en microprogramme (cas particulier de IBM 5100-5110). Ainsi l'utilisation d'APL sur l'IBM 5100 nécessite deux niveaux d'interprétation : d'abord la simulation, par le microcode, d'une machine de type 360, ensuite l'interprétation d'APL par un programme écrit dans le code de cette machine simulée (figure 4).

L'expérience montre que ce calculateur s'avère très lent dès que l'on veut traiter une application de quelque importance.

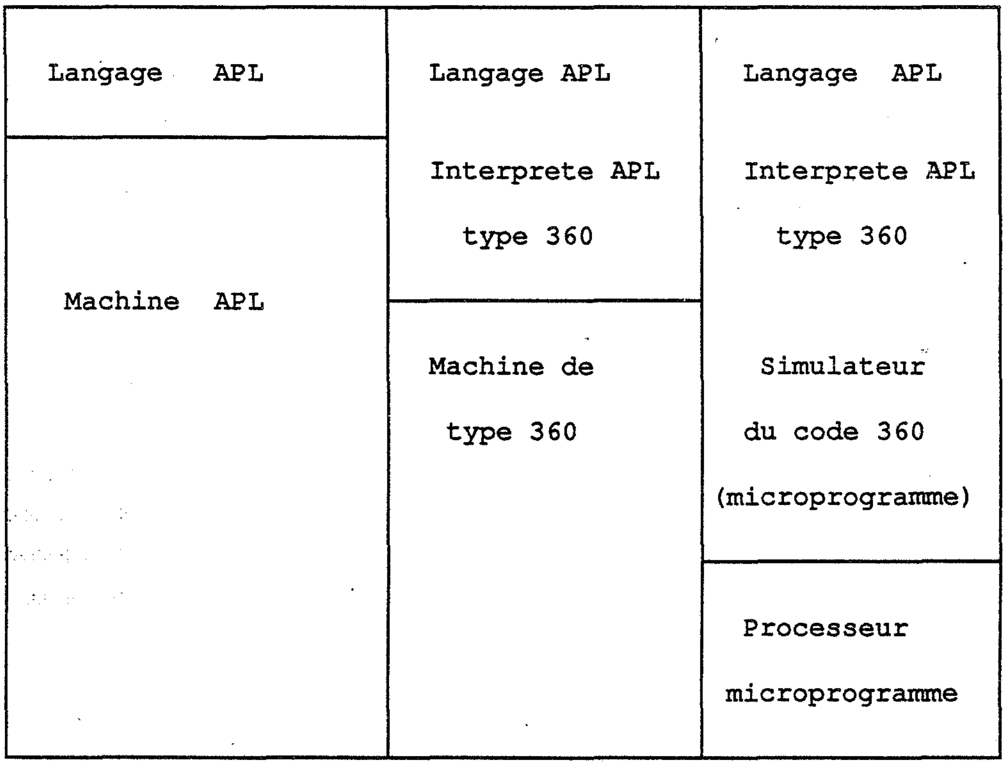
Ces architectures de machine sont, nous l'avons vu très classiques. Pourquoi n'existe-t-il pas de véritables machines APL, dont le matériel soit réellement conçu pour APL ?

En fait, (cf [Wiedmann79]) il est clair que les constructeurs d'ordinateurs sont peu enclins à concevoir un tel matériel spécialisé à moins que ce calculateur soit exclusivement dédié à APL, et que cet investissement s'avère plus rentable que tout autre étude susceptible de profiter à l'ensemble des utilisateurs.

Pratiquement, il faut reconnaitre que l'utilisation d'APL reste très marginale dans le monde de l'informatique, et ce n'est pas le semi-échec des diverses tentatives faites dans cette voie qui encouragera les constructeurs à poursuivre ce genre de recherche...

2.4 LES MULTIPROCESSEURS

Pourquoi cette approche est-elle tentante ? On pourrait penser que, grâce aux progrès de la technologie, les microprocesseurs de demain auront une puissance suffisante pour satisfaire l'utilisateur le plus exigeant. Cependant, il est probable que l'on se heurtera toujours à une application nécessitant un peu plus de puissance de calcul que disponible. Si, en revanche, on dispose d'algorithmes capables



Niveaux de realisation successifs du systeme IBM 5100

Figure 4

de faire travailler efficacement sur un problème (ici, un même programme APL) plus d'une unité centrale à la fois, on sait multiplier notre puissance de calcul, quelle que soit celle-ci. Cette approche est d'ailleurs tout à fait justifiée par la constatation qu'il est plus économique de connecter deux processeurs existants que de construire un processeur deux fois plus rapide.

L'approche visant à faire coopérer un ensemble de processeurs à une même tâche était jusqu'alors traditionnellement liée aux super-calculateurs : processeurs vectoriels ou pipe-line, tels les Cray-1, Cyber 205, etc. L'inconvénient de ces ordinateurs est que le programmeur doit penser en termes de vecteurs, alors que les langages dont il dispose ne s'y prêtent guère : ironie du sort, ces calculateurs sont principalement utilisés par des programmes FORTRAN, (langage dramatiquement scalaire s'il en fût), que l'on doit faire analyser par un préprocesseur capable de vectoriser les boucles sur les tableaux, avec l'inefficacité que l'on imagine... Il est probable, comme cela est souligné dans [Langlet82a] qu'une bonne implantation d'APL serait d'une grande efficacité sur de telles ordinateurs. Il est signalé sans autre détail dans [Langlet82a], l'existence de systèmes APL sur de gros calculateurs japonais, le Facom-380 de Fujitsu et l'ACOS 100 de Nippon Electric Company, ce dernier disposant d'un processeur vectoriel permettant l'exécution de 30 millions d'opérations flottantes par seconde...

3 L'APPROCHE D'APL 90

Pour la réalisation de notre architecture de machine, certaines techniques ont été éliminées à priori, telle une approche uniquement logicielle, ou microprogrammée, ou l'utilisation d'un matériel trop coûteux.

Une part importante des réflexions de l'équipe ayant concerné les notions d'évènements et de processus, il nous semblait intéressant, pour pouvoir mettre ces idées en oeuvre, d'opter pour une architecture multi-processeurs. En fin de compte, le créneau possible pour la réalisation était le suivant : un calculateur multiprocesseurs de faible coût doté d'un système dédié à APL acceptant une dizaine d'utilisateurs connectés simultanément.

En outre, l'idée était que, même lorsque ce calculateur ne servait qu'un seul utilisateur, le système devrait être plus performant qu'un système équivalent à un processeur unique.

Cette dernière contrainte impose une étude détaillée des possibilités de parallèlismes dans un système APL [Girardot82]. Comme nous l'avons déjà précisé cette étude ne sera pas présentée dans le cadre de ce travail. En effet elle correspond à une phase ultérieure du projet APL 90. De ce fait elle n'a pas été implémentée dans la version 1 du système APL. Nous l'avons mentionnée ici tout simplement pour donner une vision globale du projet et les incidences de certaines idées sur le choix du calculateur.

Au terme de nos réflexions sur les architectures de machine, nous avons établi le cahier des charges pour l'architecture matérielle à choisir : la machine doit disposer d'un petit nombre d'unités de traitement équivalentes (équivalent signifiant ici que les unités ne sont pas obligatoirement identiques, mais que les algorithmes utilisés doivent pouvoir être implémentés sur ces unités ; ceci ne pose théoriquement pas de problème si la programmation est faite dans un langage de haut niveau). Chaque unité doit disposer d'une mémoire locale de taille convenable, et un dispositif doit exister pour la communication entre ces unités. En pratique comme les unités n'échangeront pas que des messages de taille réduite, il devient nécessaire que ces unités puissent accéder à de la mémoire commune.

Notre choix s'est donc porté sur l'un des rares systèmes répondant à ces spécifications : la SM 90 [Finger81]. Cependant, le produit APL 90 dans sa version actuelle n'est pas liée aux spécifications matérielles de la SM 90. En effet écrit en C et sous UNIX, APL 90 a déjà été porté sur une large gamme de machines (SPS7, HP9000, etc.).

4 ARCHITECTURE LOGICIELLE D'APL 90 : ALM 90

4.1 L'UNIVERS DES OBJETS

A la différence des architectures habituelles de machines APL, ALM 90 implante un univers unique très grand, d'objets.

Cet espace des objets inclut aussi bien les temporaires, les variables de session crées par les utilisateurs, que les objets résidents, les fichiers, les zones de travail etc.

L'unicité de l'univers des objets introduit une uniformité, une simplicité et une généralité dans le traitement des données du système [Snyder79].

Un autre choix concerne la codification des noms externes en noms symboliques internes. Dans les systèmes classiques, les noms internes utilisés sont en général des numéros de symboles. Ce numéro permet d'accéder très rapidement au descripteur ou ensemble d'attributs de l'objet que le symbole désigne. C'est donc par la même occasion un numéro de descripteur. Une telle transformation évite des recherches dans la table des symboles, recherches coûteuses en temps si cette table est très grande (en particulier si cette table est située dans un espace virtuel).

Le code d'une fonction devient très dépendant du contexte en ce sens que pour utiliser la fonction dans un espace de travail différent il faudra mettre à jour tous les numéros de symboles présents dans le code de la fonction. C'est le cas notamment quand on copie dans la zone active des fonctions définies d'une zone inactive, les attributions de numéros internes aux mêmes symboles externes n'ayant aucune raison d'être identiques.

C'est pourquoi la méthode que nous proposons consiste à retarder jusqu'à l'exécution, l'édition de liens NOM <-----> VALEUR associée à ce nom. Ainsi à l'inverse des systèmes habituels, la forme interne d'une fonction ne pas fait de référence directe aux entrées de la table des symboles de la zone active. Le code interne d'APL 90 utilise des références symboliques. Ainsi le code interne d'une fonction est universel dans un espace virtuel, c'est à dire indépendant d'une table de symboles quelconque.

L'inconvénient majeur de cette méthode est que l'accès à un nom n'est plus effectué par une simple indexation dans une table mais nécessite une recherche plus complexe.

En revanche les avantages sont nombreux :

- le transfert d'une fonction d'une zone de travail (workspace) à une autre consiste en une simple copie ; **Aucune conversion** n'est nécessaire ;
- le partage d'une fonction APL entre plusieurs zones de travail devient

possible sans aucune duplication physique ;

- il devient aisé d'accéder à plusieurs tables de symboles simultanément.

4.2 INCIDENCE SUR LA GESTION MEMOIRE

La plupart des systèmes APL offrent aux utilisateurs des petits espaces disjoints d'objets. Ces espaces peuvent tenir individuellement en mémoire centrale. Une technique de recouvrement d'espaces est mis en oeuvre pour gérer les travaux en mémoire centrale.

Le concept d'univers unique d'objet ne nous permet plus d'utiliser pareille technique. Pour des raisons diverses qui sont exposées dans [Sako83], nous avons finalement adopté une gestion de mémoire hiérarchisée. Cette hiérarchie comportant deux niveaux :

- la mémoire centrale
- la mémoire secondaire

La description des aspects essentiels de la gestion mémoire trouve annexe 1.

4.3 LA REFERENCE A UN OBJET

4.3.1 LE PROBLEME

A la création d'un objet, il y a nécessité de stockage de la représentation de l'objet et de mise en oeuvre de moyens d'accès à cette représentation, autrement dit une référence à l'objet. Une référence est une chaine de bits qui identifie un objet. La détermination de la forme d'une référence est un problème assez délicat [Guiboud75]

Le problème est encore plus difficile lorsqu'on utilise une gestion mémoire à plusieurs niveaux. A tout instant la représentation doit exister en mémoire centrale ou en mémoire secondaire. Cependant un objet ne peut subir de traitement que s'il réside en mémoire centrale. Lorsqu'une opération doit s'exécuter sur un objet, on transmet à cette opération une référence à cet objet. A partir de cette référence le système doit être capable de déterminer si oui ou non l'objet est présent en mémoire centrale. Dans l'affirmative l'opération est exécutée. Dans le cas contraire, le système doit localiser l'objet en mémoire secondaire et le ramener en mémoire centrale.

Il est donc nécessaire à partir de la référence à un objet de pouvoir établir une correspondance entre les adresses en mémoire secondaire et celles en mémoire centrale. Cette opération doit être rapide.

4.3.2 NOTRE SOLUTION

Il existe dans la littérature de nombreuses méthodes de désignation des objets. Parmi elles les systèmes d'adressage par capacité ("capability systems"). Cette méthode propose l'introduction "d'adresses absolues" (au sens nom universel) dans le descripteur de l'objet.

Dans un tel système proposé par Fabry [Fabry74], chaque objet est identifié par un IDentifieur Unique (IDU) qui est une chaine de bits de longueur fixe. On assure en outre que cet identificateur soit différent de ceux des objets précédemment crées.

Deux tables sont nécessaires pour la réalisation d'une telle méthode. La première représente pour chaque objet une entrée contenant l'adresse mémoire secondaire de l'objet. La seconde table possède des entrées contenant l'adresse courante en mémoire centrale de l'objet. A chaque référence à l'objet on consulte des adresses en mémoire secondaire. Fabry propose plusieurs variantes de son système.

Sans nier les avantages d'une telle méthode nous notons que la table des adresses mémoires secondaires de tous les objets existants dans le système, risque d'occuper un espace mémoire important notamment lorsque les objets sont de petite taille. D'où une dégradation des performances du système à cause des recherches ou des accès fréquents à la mémoire secondaire.

La solution que nous proposons et qui semble mieux adaptée au système APL, intègre certaines idées de la méthode précédente. Notamment l'introduction dans la désignation de l'objet du concept "d'adresse universelle" (au sens nom universel).

En revanche nous rejetons l'implantation des tables pour les entrées de tous les objets du système. En effet certains objets APL peuvent être de taille suffisamment réduite pour que leur valeur puisse être incluse dans la référence elle même (scalaires de certains types, petits tableaux etc...). Pour les autres objets la référence qui est une chaîne de bits de longueur fixe, contiendra comme adresse universelle l'adresse mémoire secondaire où se trouve la représentation de l'objet.

En conséquence deux types de références sont à prévoir :

1- celles qui s'autodécrivent

2- celles qui contiennent une adresse mémoire secondaire.

Un bit dans la référence suffit à distinguer les deux types. Nous parlerons dans la suite de ce document de **référence universelle.**

4.4 L'ACCES AUX OBJETS

En APL certains objets sont créés dynamiquement lors de l'exécution des programmes et doivent être détruits dès lors qu'ils ne sont plus référencés. Un objet devient inaccessible dès lors qu'il n'existe plus de références de l'objet detenues par d'autres objets du système. A chaque instant l'ensemble des objets du système forme un graphe dont la racine est toujours accessible.

Comment et quand récupérer la place occupée par les objets inaccessibles ?

Plusieurs approches sont possibles. Nous retiendrons la méthode dite des **compteurs de références** dont on trouvera la justification dans [Sako83].

On associe à chaque objet un compteur qui indique le nombre "d'individus" qui connaissent l'existence de l'objet. On ne s'intéresse qu'au nombre de ces "individus" et non à leurs identités. Le compteur est incrémenté à chaque fois qu'un nouvel "individu" connait l'objet, il est décrémenté à chaque fois qu'un "individu" oublie l'objet.

Au départ l'objet est créé avec un compteur initialisé à 1, puisqu'il n'est connu que de son seul créateur.

L'objet peut être physiquement détruit dès lors que son compteur de références passe à 0, car il est devenu inaccessible.

Notons que cette méthode permet des gains importants de place mémoire. En effet elle rend possible l'optimisation des primitives APL de restructuration, le partage d'un objet par d'autres sans aucune duplication physique (cas de **A←B**) mais surtout du passage de variables comme paramètres de fonctions.

5 LES OBJETS GERES PAR ALM 90

Les objets d'ALM 90 sont les tableaux rectangulaires et les tables.

5.1 LES TABLEAUX

Les tableaux sont des arrangements suivant 0, 1 ou plusieurs dimensions, d'éléments quelconques ou **items.** Les tableaux ont les propriétés suivantes :

- éléments de type et de structure quelconques.
- accès par indices numériques.

Les types de représentation des tableaux varient suivant la nature de ces derniers :

- homogènes : c'est le cas des tableaux classiques du langage APL. Le type commun à tous les éléments est factorisé.

- progressions arithmétiques : valeurs entières ou de type flottant. Une progression arithmétique est typiquement obtenue comme résultat de la fonction iota monadique. Un tableau de type progression arithmétique est conservé sous la forme de trois nombres : valeur du premier élément, nombre total d'éléments, différence entre deux éléments consécutifs.

- hétérogène ou enclos : c'est le cas lorsqu'il n'est pas possible de factoriser le type des éléments, ou lorsqu'un item est un tableau enclos. Par exemple, cette forme est utilisée pour la représentation des fonctions de l'utilisateur.

5.2 TABLES

Les tables sont des ensembles **non ordonnés** d'éléments quelconques. Les items d'une table sont désignés par des atomes.

Dans le système APL, les tables sont utilisées comme tables de symboles ou répertoires de zones de travail. La notion de table dans le système APL 90 correspond à une formalisation de la "table des symboles", qui existe dans tous les systèmes APL, mais n'avait jamais été reconnue comme une entité à part entière.

Un élément d'une table est représenté par son nom (atome), une valeur principale (dite valeur tout court), et des valeurs secondaires, que nous appellerons **propriétés.** Ces valeurs secondaires ne sont pas utilisées par l'ALM, mais par l'interprète APL qui les utilisera pour conserver certaines caractéristiques des objets.

Il est également à remarquer qu'une table est un item, et peut donc être manipulée comme tout autre élément d'un tableau.

5.3 TYPES DE DONNEES

APL 90 permet de manipuler plusieurs types de données, numériques ou non. Voici la liste de ceux qui sont actuellement reconnus par le système :

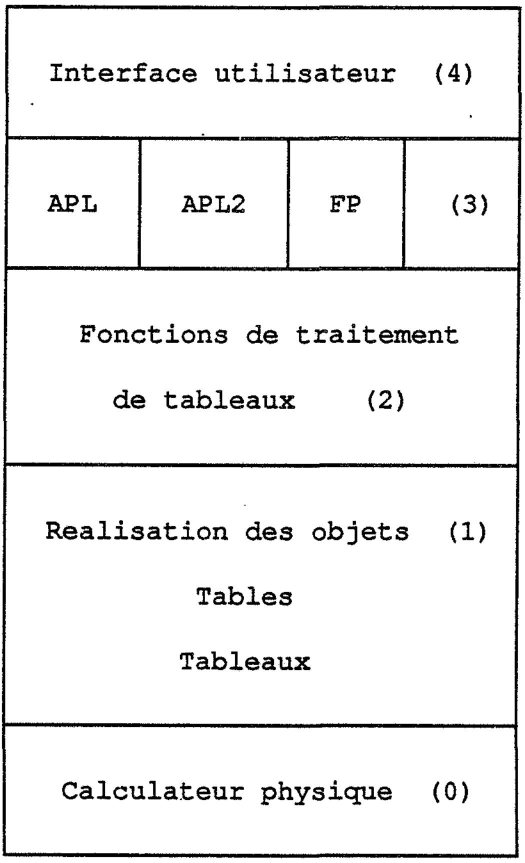
- Valeurs numériques
    - logiques 1 bit [0 1]
    - entiers 8 bits [-128 +127]
    - entiers 16 bits [-32768 +32767]
    - entiers 32 bits [-2147483648 +2147483647]
    - flottants 32 bits
    - flottants 64 bits

- Caractères
  - caractères 8 bits
    (ASCII complet + APL)
  - caractères 16 bits
  - caractères 32 bits
- Atomes
  - suite arbitraire de caractères
    {toto}
  - unité syntaxique
    + ; etc...

APL 90 permet de manière simple l'introduction de nouveaux types primitifs d'objets. Quand un nouveau type a été décrit, il peut être immédiatement manipulé par toutes les primitives de restructuration de l'interprète.

6 DESCRIPTION DE L'ALM 90

Le système APL 90 peut être considéré comme construit sur une machine virtuelle, baptisée ALM (Array Languages Machine), définie par une architecture à couches logicielles, correspondant au schéma suivant :



La couche inférieure (couche 0) correspond au calculateur physique et au système hôte.

Sur cette couche est construite la partie logicielle permettant la réalisation des objets de la machine ALM (couche 1). C'est l'univers des objets autrement dit la "mémoire" de l'ALM 90.

La couche 2 correspond aux instructions de la machine ALM, c'est à dire à l'ensemble des fonctions s'appliquant aux objets. C'est l'unité de traitement d'ALM 90. On peut y observer trois sous couches :

- la couche des fonctions de base telles que la création, la destruction, l'accès à un objet.

- la couche des fonctions primitives les plus couramment utilisées.

- la couche interface avec l'interprétation.

La couche 3 constitue la réalisation proprement dite d'un langage sur la machine ALM, qui peut être le langage APL, le langage APL2, un langage de type programmation fonctionnelle (FP), ou n'importe quel autre langage de traitement de tableaux, par exemple Nial. Son rôle est comparable à celui d'un analyseur de syntaxe.

Enfin, la dernière couche (4) représente une application conçue par l'utilisateur, et programmée dans l'un des langages disponibles. Il est d'ailleurs envisageable que cette application puisse faire appel à des modules écrits dans différents langages, par exemple APL et Nial.

chapitre 1

CHAPITRE 2

Chapitre 2

CRITERES ET EXTENSIONS

1 INTRODUCTION

Ce chapitre commence par situer certaines propositions d'extensions apparues ces dernières années dans la communauté APL. Ensuite nous établissons quelques critères auxquelles les extensions devront répondre. Les propositions de type APL2 sont analysées en premier, celles d'Iverson en second. Enfin nous discutons de l'intégration de ces extensions dans le système APL 90.

2 SURVOL DES EXTENSIONS

Les extensions que nous aborderons dans cette partie, peuvent être classées essentiellement en deux catégories :

- les types et les structures de données
- les fonctions et les opérateurs

Dans chaque catégorie, la liste des extensions que nous citons n'est pas exhaustive.

2.1 LES STRUCTURES DE DONNEES ET LES TYPES

APL reconnait seulement deux types de données : le type numérique et le type caractère, et une seule structure de donnée : le tableau rectangulaire de rang quelconque, contenant des éléments scalaires homogènes. Si cette limitation a permis l'application globale de fonctions et d'opérateurs sur les tableaux sans programmation explicite et contrôle de boucle, elle est nuisible à l'usage d'APL dans certains domaines. Les propositions d'extensions qui suivent tendent à lever ces handicaps.

2.1.1 LES FICHIERS

L'ajout des fichiers dans les systèmes APL a largement été motivé par la carence des tableaux à supporter les applications importantes orientées vers la gestion. En effet, leur structure rectangulaire et homogène est trop restrictive pour supporter facilement la représentation de structures complexes formées de données numériques et caractères. La résidence des objets APL en mémoire centrale limitait leur taille, et interdisait leur partage entre plusieurs utilisateurs[1].

Les approches consistent à permettre depuis APL l'accès soit à des fichiers classiques séquentiels, séquentiels-indexés ou directs, soit encore à des fichiers APL spécialisés à accès relatif.

Dans le premier cas, une interface avec le système de gestion de fichiers standard est introduite dans APL, ce qui permet la communication de données entre APL et les autres processeurs du système [Macon74].

Dans le second cas, un système orienté APL est créé de toutes pièces : un fichier étant considéré comme **une liste de composantes** chaque composante représentant une variable APL **quelconque.** Suivant la sophistication des méthodes d'accès, on peut lire, écrire, ajouter, détruire ou modifier des composantes [STSC73], [Juran74] ou même effectuer des compressions, expansions ou rotations sur ces listes [Pakin75]. Dans toutes les implantations, la manipulation de ces fichiers s'effectue soit par des fonctions systèmes spécialisées, soit par des nouvelles fonctions primitives, mais jamais par une extension des fonctions primitives du langage. D'autres proposent l'utilisation de variables partagées [IBM73] ce qui permet l'accès par l'intermédaire de processeurs auxilliaires spécialiés, à tous les types de fichiers gérés par le système d'exploitation.

Il est donc courant de nos jours de trouver des systèmes APL qui gèrent les fichiers. Les systèmes de fichiers de type APL*PLUS [STSC73] semblent s'être imposés aux yeux de nombreux utilisateurs. Remarquons cependant que les fichiers ne représentent pas une véritable extension aux **structures** de données d'APL mais simplement un moyen commode de concevoir une super-structure contenant des tableaux APL.

2.1.2 LES ARBRES

Admettre dans APL une structure de donnée de type arbre a nécessité la définition de nouvelles fonctions primitives pour la manipulation des noeuds et des feuilles [Alfonseca76, Murray73] mais il faut reconnaitre que ces extensions n'ont pas eu beaucoup d'adhésion ces dernières années à cause de la focalisation de l'attention des chercheurs autour des tableaux enclos qui permettent une généralisation des structures existantes[2].

---

[1] Ceci n'est plus tout à fait vrai aujourd'hui avec les systèmes à mémoire virtuelle.

[2] Essentiellement semble t-il pour des questions de termonologie, l'introduction de la notion d'arbre semblant inquiéter plus l'utilisateur que la généralisation des tableaux, car de fait les fonctionnalités proposées sont très voisines.

2.1.3 LES TABLEAUX GENERALISES

C'est la solution la plus importante pour lever la limitation sur la structure des tableaux APL. L'idée est de pouvoir créer des structures plus complexes et plus générales que les tableaux rectangulaires, denses et homogènes d'APL-ISO.

Les tableaux généralisés ont constitué un des plus grands centre d'intérêt pour la communauté APL. Les propositions d'extension ont longtemps relevé d'une bataille d'experts [Ghandour73, Gull79, Alfonseca76, Edwards73, Murray73, Vasseur73, Bernecky80, More79, Brown71, Smith81]. Les divergences fondamentales concernent les définitions de nouvelles fonctions primitives qui sont indispensables pour manipuler les tableaux généralisés et plus particulièrement l'effet de ces fonctions sur les scalaires simples (c'est à dire non enclos). Ces nouvelles fonctions sont : **enclose** qui s'applique à un tableau quelconque pour en faire un scalaire enclos qui peut devenir élément d'un autre tableau, et **disclose** qui extrait les éléments enclos d'un tableau généralisé.

L'une des pierres d'achoppement est la notion de **tolérance** liée à la fonction **enclose,** qui détermine si le résultat de la fonction enclose appliquée à un scalaire simple est -ou non- identique à ce scalaire. Ce choix détermine l'ensemble des comportements des tableaux face à cette généralisation.

Remarquons que le besoin de tableaux généralisés avait donné naissance à des organisations de données très diverses, telles :

- les packages de SHARP APL [Berry79]
- les arbres d'APL-LAVAL [Robichaud77]

Ces organisations demeurent cependant des organisations de **noms.**

Les tableaux généralisés apportent à l'utilisateur la possibilité de créer et de manipuler des **données de structure arborescente.** On retrouve en un sens les "records" de COBOL ou de PASCAL. Si la réalisation à laquelle cette innovation s'intègre supporte une mémoire virtuelle, elle ouvre la porte aux fichiers et plus généralement aux bases de données [Nakache76]. Il manque cependant la possibilité d'accès concurrents.

De nombreuses réalisations sont décrites dans la littérature :

- Simulation d'un système de fichiers à accès direct organisés en liste de composantes [Edwards73] ;

- Réalisation du système de base de données relationnelles G.E.S.O.P. à l'aide de tableaux généralisés [Pierre79].

Les tableaux généralisés trouvent également des utilisations dans les applications graphiques ou en analyse de langages [Gull76].

Comme illustré dans [Bertin81], les tableaux généralisés résolvent un problème technique posé par le langage APL ; le nombre limité de paramètres d'une fonction définie, par la possibilité de regrouper les paramètres disparates en une seule variable. Ceci permet par exemple de résoudre des problèmes d'interface entre APL et d'autres langages comme Fortran.

Après douze années de débat et de bataille de chapelles, l'extension d'APL aux tableaux généralisés (generalized arrays, nested arrays, enclosed arrays) aboutit de nos jours à un manque de consensus et de compromis entre les différentes tendances.

Les temps forts de ce long débat sont rapportés dans [Baron82].

Nous constatons que les difficultés se sont exprimées essentiellement à deux niveaux :

- le nombre important de questions soulevées par les tableaux généralisés qui sont restées sans réponse, ou qui admettent des réponses multiples ;

- la difficulté d'allier la simplicité à la généralité dans la conception et la réalisation des extensions.

Ainsi on se trouve aujourd'hui avec une norme ISO d'APL, qui exclut bien sûr les tableaux généralisés, et un certain nombre de systèmes expérimentaux ou d'implantations d'APL qui offrent des extensions incluant ces tableaux généralisés.

Il existe en fait deux implantations majeures : le système APL2 (IBM) et le système IPSA (I.P. Sharp Associates Limited) qui illustrent les idées des deux chapelles. L'une tenue par Jim Brown (IBM) et Bob Smith (STSC) et l'autre par K.E. Iverson et Bob Bernecky (IPSA).

2.1.3.1 Fonctions et opérateurs pour tableaux généralisés Plusieurs propositions existent en plus des deux fonctions enclose et disclose. Citons en quelques unes :

- **Fonctions de modifications de niveau :** puisqu'un tableau peut être enclos et cela de manière récursive, nous appellerons **profondeur de capture** le nombre de niveaux d'inclusion qui sépare le tableau et les scalaires simples.

Ainsi pour des facilités de manipulation des tableaux enclos, il est commode de disposer de fonctions permettant de spécifier le niveau de profondeur de capture du tableau ou de ses éléments, sur lequel les fonctions doivent s'appliquer.

- **Conversion structure rectangulaire-tableaux enclos :** Ces fonctions permettent le passage d'un tableau rectangulaire (donc ordonné suivant des axes) à un tableau enclos (ordonné suivant une profondeur) et vice-versa.

- **Les opérateurs de profondeur :** Ils permettent l'application d'une fonction à chaque item d'un tableau enclos.

- **L'indexation généralisée :** C'est la possibilité d'admettre comme indice d'un tableau, des tableaux eux-mêmes enclos.

2.1.3.2 Type - Structure vide - Elément de remplissage Dans APL, les tableaux ne peuvent être que de type numérique ou caractère. Qu'en est-il des tableaux enclos, surtout s'ils peuvent être hétérogènes ? Les propositions concernent aussi la représentation des tableaux vides enclos et la définition d'élément de remplissage à utiliser lorsqu'un tableau enclos nécessite d'être complété.

2.1.3.3 La pénétrabilité : il s'agit d'admettre que les fonctions primitives scalaires soient pénétrantes, c'est-à-dire qu'elles s'appliquent récursivement à tous les niveaux "d'inclusion" jusqu'à atteindre les scalaires simples.

2.1.3.4 Entrée et sortie des tableaux enclos En entrée, une controverse concerne ce qu'on appelle la "strand notation". D'après cette notation, une séquence de tableaux juxtaposés est interprétée comme un vecteur d'éléments enclos. L'argument essentiel est qu'elle est similaire à la notation pour les vecteurs de constantes dans APL-ISO. Les adversaires soutiennent que la "strand notation" n'est pas une vraie extension de la notation vectorielle dans la mesure où elle respecte pas certaines propriétés de cette dernière [Falkoff81].

En sortie, les propositions concernent essentiellement la visualisation des tableaux enclos.

2.1.4 NOMBRES COMPLEXES

Il s'agit d'ajouter un type **numérique complexe** de manière explicite. Ce qui soulève de nombreux problèmes : extension des domaines des fonctions primitives, représentation, conversion ... [McDonnel73, Forkes81].

2.1.5 TYPES DEFINIS PAR L'UTILISATEUR

Le but est de fournir à l'utilisateur le moyen de créer de nouveaux types de données autres que numériques et caractères, qui seront d'usage plus naturelles dans le contexte d'une application. Certains langages inspirés d'APL comme X\APL ou ALICE [Jenkins80] offrent ces facilités. Un des aspects importants de la seconde partie est justement la définition de type par l'utilisateur.

On peut également citer les travaux de L.P.A. Robichaud de l'université de Laval au Québec qui a publié en 1973 une intéressante extension d'APL appelée *\APL ("Power Extension of APL") dont le principe est le suivant :

- des objets non APL peuvent être introduits grâce à une primitive nouvelle appelée ionisation : on modifie un bit dans la table des symboles afin que l'objet provoque une erreur lorsque l'interpreteur le rencontre,
- un puissant mécanisme de gestion d'interruptions permet d'avoir accès à tout le contexte de l'interpreteur au niveau de l'interruption.

Ce qui a permis des extensions vers le calcul symbolique et la manipulation d'arbres (structures).

2.2 FONCTIONS ET OPERATEURS

Les fonctions et les opérateurs d'APL sont simples et puissants. A l'heure actuelle les opérateurs ne peuvent pas être définis par l'utilisateur. Ce sont des primitives du système. Leur rôle est de fournir des fonctions dérivées à partir d'arguments. Ces arguments sont obligatoirement des fonctions primitives scalaires sauf pour / qui admet également des tableaux.

Les fonctions définies quand à elles ont une valence fixe, c'est à dire qu'elles sont soit monadiques soit diadiques, contrairement aux fonctions primitives qui elles peuvent être ambivalentes.

Les extensions proposées consistent à uniformiser le concept d'opérateur, à admettre comme arguments d'autres fonctions ou des tableaux et à lever les restrictions sur les fonctions définies.

3 CRITERES POUR DES EXTENSIONS D'APL

Les extensions proposées dans les domaines cités ci-dessus présentent des intérêts d'importances inégales. Elles ne sont pas toujours indépendantes les unes des autres. Par conséquent, il est impossible de les étudier de manière isolée. De même, juger de l'opportunité d'une extension est assez délicat, si on ne se définit pas certaines règles.

Pour celà, il faut avoir à l'esprit les aspects d'APL qui ont rendu le langage jusqu'à présent attratif et facile à utiliser. On peut citer **la simplicité de la syntaxe** des expressions, **l'application globale** des fonctions sur les tableaux, la **généralité** de certains concepts, **le comportement prévisible** des opérations dans les **cas limites, les identités** (ou équivalences) qui fournissent une description formelle et précise des opérations [Falkoff73a].

3.1 QUALIFICATIONS D'UNE EXTENSION

Toute extension, quel que soit son intérêt ou son domaine d'application, peut se voir attribuer l'un des trois qualificatifs ci-après définis par J.A. Brown [Brown79].

a. Extension "propre"

C'est l'extension d'une opération existante. Une extension propre ne nécessite pas la reprogrammation des fonctions définies utilisant la dite opération. Toutes les expressions qui fonctionnaient avant l'extension doivent fonctionner après. C'est par exemple le cas de la concaténation (, diadique) qui fut étendue aux tableaux de rang supérieur à 1, l'effet de cette fonction sur les vecteurs et les scalaires restant inchangé.

b. Raffinement

Le raffinement d'une opération existante altère son comportement précédent l'extension. Lorsque la fonction primitive représentation (⊤ diadique) fut étendue aux tableaux de rang >1, sa définition sur les vecteurs à un élément fut modifiée pour des raisons de cohérence. La même remarque peut être faite pour la fonction de transposition monadique (⍉)

Le raffinement peut donc entrainer la ré-écriture des fonctions définies utilisant l'opération concernée.

c. Addition ou ajout

Toute extension qui ne constitue pas la modification d'une opération existante est l'addition d'une nouvelle possibilité au langage ou au système.

Les fonctions **enclose** (⊂) et **disclose** (⊃) sur les tableaux généralisés sont de ce type. Une nouvelle extension ne nécessite pas bien entendu la modification des fonctions définies, mais certaines applications pourront tirer une certaine efficacité, ou une plus grande concision, en utilisant ces nouvelles opérations.

3.2 CRITERES

L'adoption d'une extension peut être guidée par plusieurs principes, et très souvent le choix final relève d'un compromis. C'est pourquoi la conception d'un langage est plus un art qu'une science, et il est fréquent que partant d'un même problème l'on arrive à des conclusions différentes.

Une notion importante, commune aux quatre critères qui suivent est la **notion d'équivalence** ou **d'identité.** En effet elle constitue la clé de toute conformité, elle permet de rendre les extensions simples et compréhensibles donc facilement utilisables.

Par exemple le résultat de la fonction d'indexation serait difficile à imaginer sans la règle d'équivalence :

```
ρM[I;J]↔ (ρI) , (ρJ)
```

Il nous parait important de préserver les règles d'équivalence partout où cela est possible.

3.2.1 LA COMPATIBILITE

La compatibilité est la mesure de l'étendue des changements qu'une proposition d'extension impose dans le langage ou le système APL. Il y a quatre sortes de changements qui affectent la compatibilité :

1. les expressions erronées qui deviennent valides

2. les expressions correctes qui fournissent du fait de l'extension une réponse différente d'APL standard.

3. les expressions correctes qui deviennent erronées

4. les expressions erronées qui fournissent des messages d'erreur d'une autre nature.

Le cas 1 n'est pas un évènement très important car c'est l'essence même d'une extension que de fournir une réponse là où ce n'était jusqu'alors pas possible. Ce phénomène a toujours été vécu dans l'histoire du langage APL.

Les extensions qui font qu'une expression fournit une réponse différente de celle d'APL-ISO (cas 2) sont d'une importance capitale pour la compatibilité et ne doivent être retenues que pour des raisons fondamentales.

Les modifications qui entrainent des messages d'erreur pour des expressions d'APL-ISO valides (cas 3), apparaissent importantes au premier regard alors qu'elles ne le sont pas en réalité. En effet, le système APL permet de situer de manière précise l'erreur ainsi que sa nature. Donc il est facile à l'utilisateur de corriger ces expressions.

Le cas 4 n'a pas d'incidence sur la compatibilité mais affecte la documentation du système ou l'expérience de l'utilisateur.

3.2.2 LE FORMALISME

Il mesure le degré de conformité d'une proposition d'extension aux règles du langage, et permet donc de déterminer si une extension est consistante (c'est à dire si elle en accord avec la théorie des tableaux [More79]).

Une extension formellement incorrecte ne doit jamais être adoptée. Cependant, il peut exister plusieurs formalismes corrects pour une même proposition, le choix final doit être guidé par d'autres principes.

Les arguments formels peuvent se décrire normalement en termes d'équivalence.

Une notation étendue et formellement correcte doit être choisie de manière à être facilement compréhensible, et, si possible, des règles d'équivalence, universellement vraies, doivent décrire la nouvelle fonction.

Exemple : la nouvelle fonction primitive **disclose** (⊃) est définie dans certaines propositions comme l'inverse gauche de la fonction **enclose** (⊂) :

```
A ↔ ⊃⊂A
```

Cette identité est préservée même lorsque l'argument est non scalaire.

3.2.3 LA SIMPLICITE

Ce critère peut se formuler de diverses manières :

- Avoir le minimum de règles pour une extension
- Traiter des classes d'objets qui ont des propriétés similaires plutôt qu'un ensemble de cas isolés
- Fournir tant que faire se peut des résultats qui soient conformes aux arguments
- Définir des règles qui soient les plus générales possibles.

Ainsi, il est reconnu que le fait que l'ensemble des fonctions dites scalaires s'appliquent toutes de manière identique sur leurs arguments est un gage de grande simplicité.

3.2.4 LE CONFORT D'UTILISATION

Il reflète la facilité de compréhension et d'application d'une notation. C'est un critère difficile à mesurer tant des considérations subjectives entrent en jeu.

Notons cependant que le confort d'utilisation d'une extension est largement influencé par son formalisme et sa simplicité.

Une preuve de confort d'utilisation est la capacité de prévoir la manière dont travaille une opération dans une situation non coutumière : c'est la loi de la moindre surprise.

3.3 REMARQUE

Les principes qui décident d'une extension ne sont pas toujours indépendants les uns des autres. Il est par conséquent tentant de classer les critères ci-dessus dans un ordre prioritaire. Le formalisme est peut être plus important que la compatibilité, le confort d'utilisation que la simplicité. Lorsqu'une extension ne répond pas à un de ces critères que faire ? A notre avis, la réponse à cette question se trouve dans sa compatibilité avec APL-ISO.

Dans la suite de ce chapitre, ainsi que dans la seconde partie nous tenterons de mettre en oeuvre ces principes.

4 LES EXTENSIONS DE TYPE APL2

4.1 LES TABLEAUX GENERALISES

Un **tableau généralisé** est un tableau au sens APL, dont les éléments ne sont plus uniquement des **scalaires** (nombres ou caractères) mais peuvent être d'autres **tableaux.**

Ainsi un **tableau généralisé** conserve une **structure rectangulaire** classique. Cette structure est définie par la connaissance **d'un rang, d'un vecteur de dimension** (s) et d'un ensemble ordonné de valeurs : ses **éléments.** Un élément d'un tableau généralisé est soit un tableau simple, soit un autre tableau généralisé.

En conséquence, les tableaux simples ne peuvent se retrouver qu'aux **feuilles** des arborescences ainsi définies.

On garde la structure de **tableaux de scalaires** (au sens APL standard) mais on offre un moyen d'enrichir la notion de scalaire par celle de structure grâce à une fonction primitive appelée **ENCLOSE.** Dès lors un scalaire peut-être un scalaire de base (nombre ou caractère) ou un **scalaire enclos** c'est à dire qu'il **peut cacher une structure complexe.**

Pour découvrir les structures cachées (s'il y en a), on dispose d'une fonction inverse de l'enclose, appelée **DISCLOSE.**

Nous utiliserons à partir de maintenant une notation sous forme de boite pour représenter les tableaux généralisés.

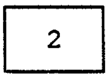
Ainsi :

- un scalaire sera visualisé comme une boîte

```
A←2
S←'X'
```

donneront :

A **S**

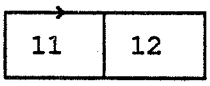


- Un vecteur sera représenté par une suite de boites accolées, une flèche horizontale sur le bord supérieur symbolisant l'arrangement des éléments suivant un axe unique.

```
B←11 12
C←'ABC'
```

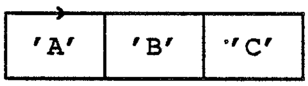
donneront :

B



11	12

C



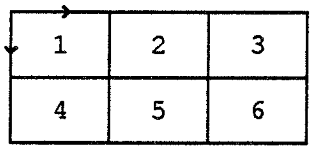
'A'	'B'	'C'

- Une matrice sera un arrangement rectangulaire de boîtes avec deux flèches indiquant les axes du tableau

```
M←2 3 ⍴⍳6
D← 2 2 ⍴'ABC'
```

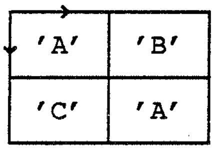
donneront :

M



1	2	3
4	5	6

D



'A'	'B'
'C'	'A'

Notons déjà que APL2 admet des **tableaux hétérogènes,** Ainsi en APL2 on peut mélanger des données numériques ou caractères.

Dans un souci de clarté et de structuration, nous n'allons pas détailler ici la panoplie des extensions qu'induisent l'adoption des tableaux. En effet, ces extensions induites rentrent dans la classification que nous avons définie (extensions propres, raffinement, ajout) et affectent soit les fonctions primitives, soit les fonctions définies, soit les opérateurs. Par ailleurs elles ont des incidences sur certains aspects du langage comme les tableaux vides ou les éléments de remplissage des tableaux.

C'est pourquoi elles seront introduites au fur et à mesure que nous aborderons les points cités. Ceci permet à notre sens de mieux saisir les motivations et les concepts qui ont présidé à leur acceptation.

4.2 LES FONCTIONS

L'adoption des tableaux généralisés entraine une avalanche de besoins qui vont à leur tour exiger la définition de nouvelles fonctions ou l'extension de certaines du langage APL.

4.2.1 Définitions de nouvelles fonctions primitives

4.2.1.1 Fonction monadique de capture La fonction dite ENCLOSE permet la **représentation scalaire** d'une structure. Elle a pour effet "d'encapsuler", "d'enfermer" ou "de recouvrir" (enclose, seal, conceal). Elle s'applique à une donnée quelconque, pour créer un scalaire généralisé dont l'unique valeur est cette donnée.

La fonction enclose est notée ⊂ dans APL2.

En APL, un scalaire se caractérise par le fait qu'il est de rang 0.

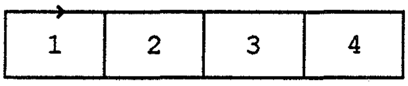
```
      ρρ5
0
      V←ι4                                  M←2 3 ρι6
      ρρV                                   ρρM
1                                     2
```

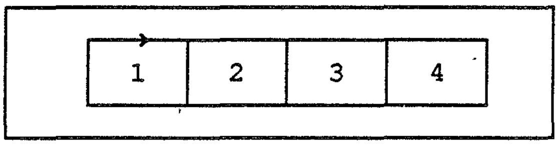
Si on veut assurer une représentation scalaire de ***V*** ou de ***M*** il suffit d'écrire :

```
       SV←⊂V                                 SM←⊂M
       ρρSV                                  ρρSM
0                                     0
```

La représentation sous forme de boites de ces objets est :

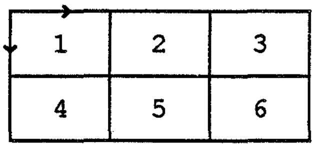
V ***SV***

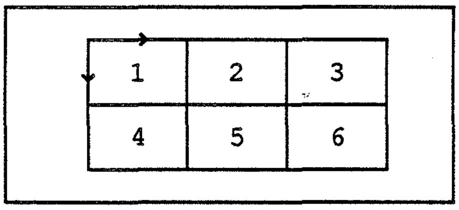




M

SM





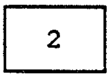
APL2 propose que l'enclose d'un scalaire de base (ou scalaire simple) soit identique à ce scalaire de base.

Autrement dit si S est un scalaire de base :
S est équivalent à ⊂*S*

S←2

S

⊂*S*



Cette proposition a d'importantes conséquences, comme nous le verrons, dans l'extension de certaines fonctions ou opérateurs d'APL.

4.2.2 Fonction monadique de libération

C'est la primitive monadique DISCLOSE, elle assure la fonction inverse de la précédente. Elle permet de dégager un niveau de capture. Elle est notée ⊃ dans APL2.

Notons que dans APL2 un scalaire de base ne cache pas de structure.

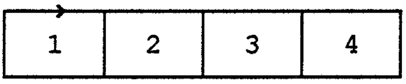
Ainsi le scalaire 2 est identique à DISCLOSE 2 :

2

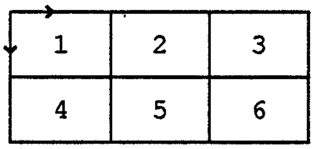
⊃2

DISCLOSE est la fonction inverse d'ENCLOSE, et révèle l'objet contenu dans le scalaire enclos.

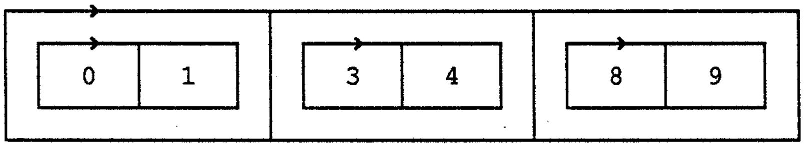
⊃*SV*



⊃*SM*

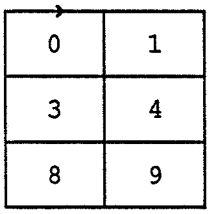


W



W est un vecteur généralisé de 3 éléments. Chaque élément étant lui-même un vecteur à deux éléments.

⊃***W*** ρ***W*** ρ⊃***W***



```
3
3 2
```

Ainsi ⊃ s'applique à toutes les composantes de la structure créant une dimension supplémentaire.

4.2.3 Remarque

L'usage des tableaux généralisés a rapidement fait apparaître des besoins en primitives qui permettent par exemple de connaître la profondeur d'imbrication d'un tableau, son type ou son élément de remplissage (parfois appelé prototype), etc ...

4.2.4 Extensions des fonctions primitives scalaires

Les fonctions scalaires en APL s'appliquent à des données **scalaires** et produisent un résultat **scalaire.** Lorsqu'elles s'appliquent à des données non scalaires, l'opération est réalisée élément scalaire par élément scalaire.

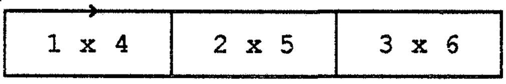
Ainsi :

```
      3 × 2
6
```

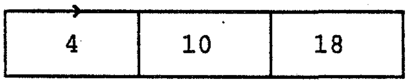
mais

```
      1 2 3 × 4 5 6
```

applique × élément par élément et fournit :

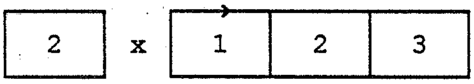


ce qui donne



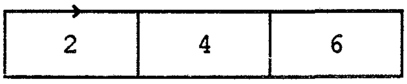
Cela nécessite implicitement que les arguments aient les mêmes dimensions. Cependant si l'un des arguments est scalaire il est étendu à la dimension de l'argument non scalaire.

Ainsi :



est évalué comme

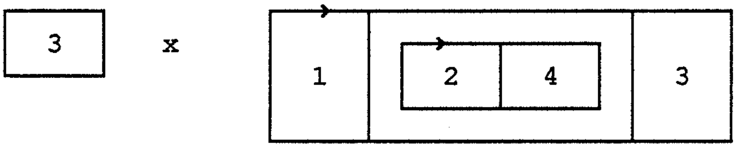
et fournit comme résultat



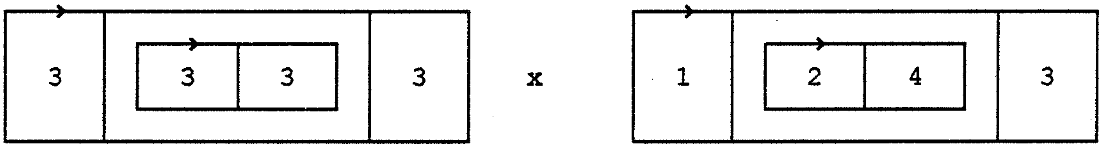
L'introduction des tableaux généralisés nous place dès lors devant une alternative :

1 - Faire la même analyse que dans APL standard. Etendre quand cela est possible (homogénéité de type des feuilles des structures en présence) l'action des primitives scalaires aux nouvelles structures. Ceci suppose donc une généralisation du principe d'extension des scalaires.

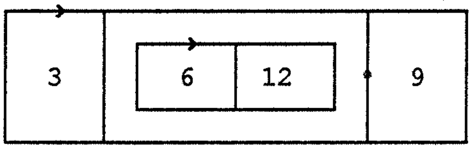
Ainsi :



est traité comme

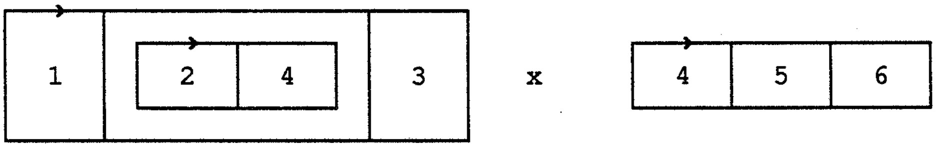


et fournit comme résultat

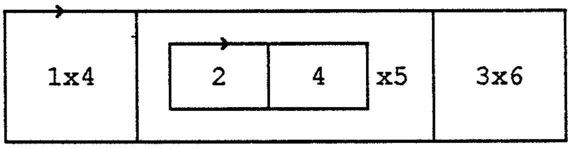


L'opération s'est appliquée à toutes les feuilles. Le scalaire 3 a été étendu en "profondeur" à la structure de droite.

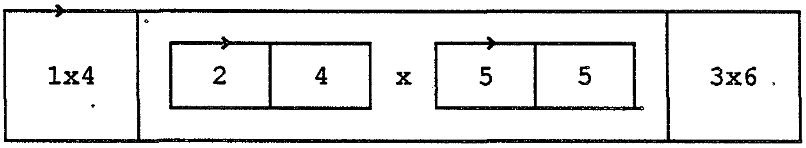
Mais l'idée va plus loin encore. Dans :



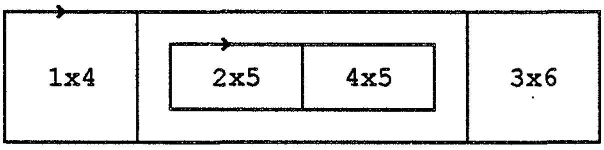
la multiplication s'applique élément par élément et donne :



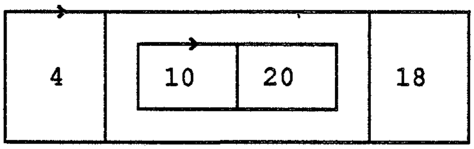
dans le deuxième élément, il y a extension du scalaire 5, soit donne :



d'où :



le résultat est :



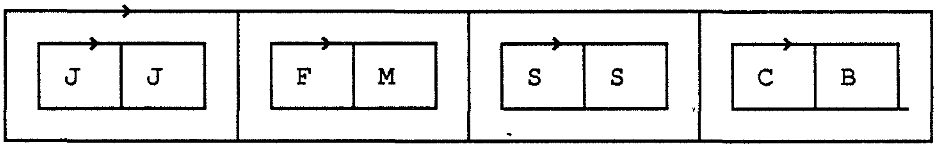
Ce choix implique un usage récursif de la définition des fonctions scalaires sur les tableaux. Dans le cas des tableaux enclos, cette récursion persiste tant qu'on n'atteint pas des scalaires simples. On dit que les fonctions scalaires sont **pénétrantes.** C'est la solution qui est adoptée par APL2.

2 - Introduire de nouveaux opérateurs qui permettent d'appliquer explicitement les fonctions au premier niveau des tableaux ou à tous les niveaux de manière récursive. Cette extension présente plus de généralité et englobe la précédente. Nous reviendrons sur ce choix dans le paragraphe concernant **les opérateurs de composition** d'Iverson.

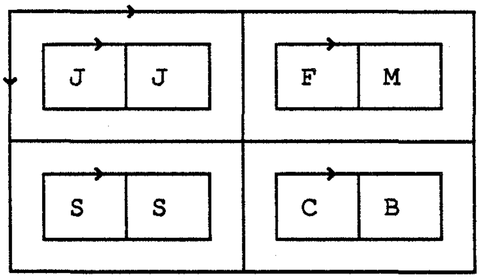
4.2.5 Extension des fonctions primitives de restructuration

Pour des commodités de construction il est utile d'étendre la fonction de concaténation (, diadique) et de restructuration (⍴ diadique). Il devient dès lors possible de construire des vecteurs de tableaux de tableaux :

```
      A←'JJ'
      B←'FM'
      C←'SS'
      D←'CB'
      APL←(⊂A) , (⊂B) , (⊂C) , (⊂D)
      ⍴APL
4

                    APL
```



```
        MAPL
```



```
        ρMAPL
2 2
```

De même il existe diverses propositions pour étendre toutes les fonctions primitives de sélection de manière uniforme. Si leur extension semble triviale, certaines comme la primitive **prend** (↑ diadique) posent quand même le problème du remplissage complétif.

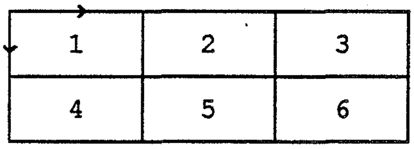
En APL, il existe deux types de scalaires simples : numérique et caractère. Les éléments de remplissage sont 0 pour les nombres et le "blanc" pour les caractères.

Ainsi

```
      4 ↑ 2 3 4                    4 ↑ 'ABC'
2 3 4 0                      'ABC '
```

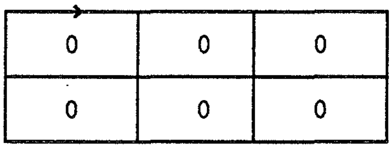
Ce choix dans APL est arbitraire mais pratique. Pour les tableaux généralisés certains préconisent de compléter par des objets vides. D'autres proposent que le type d'un tableau soit un tableau de même structure dans lequel on remplace chaque élément scalaire des feuilles par un "blanc" ou un 0 selon qu'il est numérique ou caractère. Donnons quelques exemples :

```
      M← 2 3 ρ ι6
      M
```



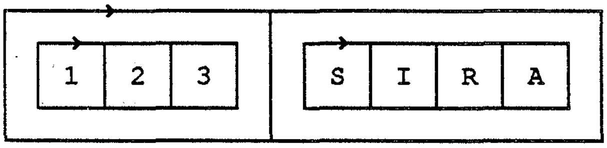
1	2	3
4	5	6

```
TYPE DE M
```



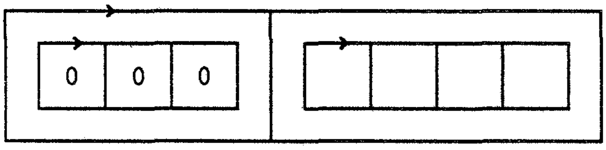
0	0	0
0	0	0

```
      X ← (⊂1 2 3),(⊂'SIRA')
      X
```



1 2 3	S I R A

```
      TYPE DE X
```



0 0 0	

Dès lors, on peut définir pour un tableau le type de son premier élément : on parlera de **prototype.** Chez APL2 le premier élément est obtenu par la fonction (↑ monadique).

```
      ↑M                        ↑X
1                         1 2 3
```

Le prototype de M ou de X est alors

```
   PROTOTYPE DE M                PROTOTYPE DE X
0                              0 0 0
```

La fonction ↑ est étendue pour fournir, appliquée à un objet vide, le prototype de cet objet :

```
      ↑ι0
0
```

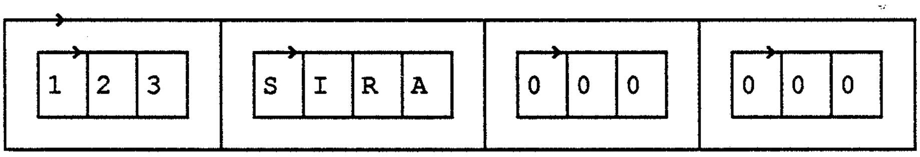
Si *T* est un tableau, on peut établir les équivalences suivantes :

```
TYPE DE T ↔ ↑0⍴⊂T
PROTOTYPE DE T ↔ ↑0⍴⊂↑T
```

Ainsi de manière générale on définira **l'élément de remplissage** d'un tableau simple ou non, comme étant son **prototype.**

D'où :

```
4 ↑ X
```



Ce choix est tout aussi arbitraire que dans le cas APL. Notons que la notion de prototype est délicate à definir pour les tableaux vides[3].

4.2.6 Indexation généralisée

L'indexation doit être étendue aux structures arborescentes que permet l'enclose. L'objectif est de permettre d'atteindre n'importe quel niveau en précisant simplement quel est le chemin pour le désigner ou l'extraire.

L'indexation étendue est réalisée en APL2 par la fonction PICK. Elle est représentée par le DISCLOSE diadique de la manière suivante :

```
I ⊃ T
```

ou *I* représente un chemin dans le tableau *T*.

Dans APL l'indexation présente une anomalie du point de vue de la syntaxe du langage. C'est une seule fonction dont la représentation nécessite deux symboles ([et]) englobant une expression APL de longueur quelconque, et elle utilise le point-virgule (;) comme séparateur non fonctionnel. Par ailleurs la liste des indices entre crochets n'est pas un tableau APL, et n'est manipulable par aucune primitive du langage.

---

[3] Nous suivrons APL2, qui définit le prototype d'un tableau vide comme étant le prototype du tableau à partir duquel ce tableau vide a été obtenu.

L'extension de l'indexation résout une partie de ces problèmes.

```
M← 3 3 ρι9
M
```

```
┌→──┬───┬───┐
↓ 1 │ 2 │ 3 │
├───┼───┼───┤
│ 4 │ 5 │ 6 │
├───┼───┼───┤
│ 7 │ 8 │ 9 │
└───┴───┴───┘
```

```
M[3;1]
```

```
┌───┐
│ 7 │
└───┘
```

peut s'écrire :

```
(⊂3 1)⊃M
```

```
┌───┐
│ 7 │
└───┘
```

Dès lors on remarque que si on pose :

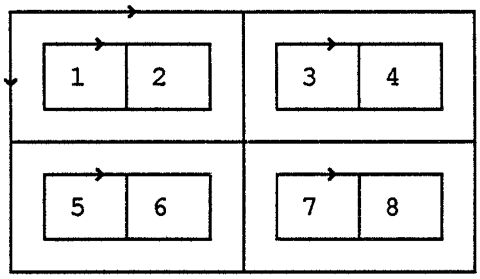
```
I←⊂ 3 1
```

alors

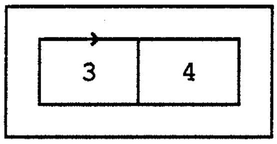
```
M[3;1]↔ I⊃M
```

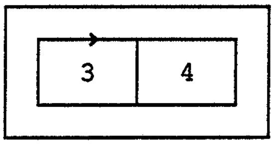
Ainsi la liste des indices (3;1) devient un tableau APL ; et mieux encore, il devient possible de passer une telle liste d'indices comme paramètre d'une fonction.

```
T
```



```
      T[1;2]                        (⊂1 2)⊃T
```



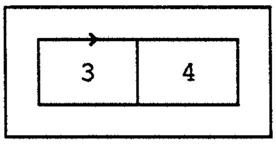


Si on pose :

```
      I←⊂1 2
```

on a :

```
      ⍴I                  ⍴⍴I
                        0
   I⊃T
```

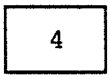


Si on pose :

```
      J←(1 2) 2
```

on a :

```
      ⍴J                  ⍴⍴J
2                       1
   J⊃T
```



Les opérations d'indexation acquièrent ainsi plus de généralité et surtout leur formulation devient indépendante du rang de l'objet indexé.

4.2.7 FONCTIONS DEFINIES

Les fonctions définies par l'utilisateur en APL s'utilisent de la même manière que les fonctions primitives du langage : elles sont appelées en plaçant leur nom dans une expression, entre ou devant leurs arguments, la valeur de leur résultat est retournée au point d'appel.

Malheureusement, il existe au moins deux cas où les fonctions définies ne se comportent pas comme des fonctions primitives APL : elles ne peuvent pas être **ambivalentes** et ne peuvent pas être arguments d'un opérateur.

L'extension proposée consiste à admettre, comme pour les fonctions primitives, qu'un même nom de fonction puisse représenter à la fois la forme monadique ou diadique d'une fonction définie, le contexte permettant de savoir quelle est la forme adéquate à mettre en oeuvre.

D'autre part, pour qu'une fonction définie puisse être argument d'un opérateur dans APL , il faudra élargir le domaine d'application des opérateurs aux fonctions mixtes, car les fonctions définies ont habituellement un comportement qui les apparente aux fonctions mixtes.

En APL2, on utilise les tableaux généralisés pour définir les résultats de l'application des opérateurs aux fonctions mixtes.

4.3 LA REPRESENTATION VECTORIELLE

C'est une notation qui fut introduite par T. More en 1973. Elle se définit de la façon suivante : si A, B, C, D, E sont cinq variables alors

```
A B C D E ↔ (⊂A) , (⊂B) , (⊂C) , (⊂D) , (⊂E)
```

Cette notation qui a eu plusieurs dénominations (*strand notation, vector notation*) a donné lieu à des controverses assez vives illustrées par une correspondance [Iverson-Brown-Smith81] échangée entre K. Iverson (IPSA) B. Smith (STSC) et J. Brown (IBM).

T. More donne la définition suivante [More73] :

"Une *Strand* est une chaîne de caractères composée d'une ou de plusieurs variables, des numériques, des chaînes entre apostrophes, des termes enclos entre parenthèses, et des constantes séparées par des espaces. Un espace adjacent à une parenthèse peut être omis. Les *strands* dénotent des listes ... L'opération de concaténation implicite dans la formation d'une "strand" est effectuée avant toute autre opération."

Trois types de constructions apparaissent dans la "strand notation" :

- les valeurs numériques telles que 12
- les vecteurs de constantes caractères tels que

`'TOMBOUCTOU'`

- les expressions tels que

`(2×C)`

Lorsque deux ou plusieurs de ces formes sont adjacentes, chacune est interprétée comme étant un item. Les résultats des constructions qui évaluent des scalaires simples restent simples.

La "strand notation" apparait ainsi comme une extension de la notation courante des vecteurs de constantes numériques. Un élément pouvant être un nombre (comme dans APL), un scalaire ou vecteur caractère, ou une expression. Une expression doit être entourée de parenthèses afin de limiter sa portée.

`12 'TOMBOUCTOU' (2×C)`

dénote un vecteur de 3 éléments enclos.

Iverson dans sa critique de la "strand notation" utilise les fonctions **Link** et **Pair** suggérées par More et dénotées par (;̃) et (;̄).

Link:

`A;̃B ↔ (⊂A),B`

Pair:

`A;̄B ↔ (⊂A),⊂B ↔ A ;̃⊂B`

Ainsi on a les équivalences suivantes :

```
A B C D E ↔ (⊂A),(⊂B),(⊂C),(⊂D),⊂B
          ↔ A;̃B;̃C;̃D;̃⊂E
          ↔ A;̃B;̃C;̃D⍮E
```

```
A (B C) D E ↔ A;̃B(;̃C);̃D⍮E
A (B (C (D E))) ↔ A⍮B⍮C⍮D⍮E
(1 2) (3 4) (5 6) (7 8) ↔ 1 2;̃ 3 4;̃ 5 6;̃ 7 8
```

Iverson propose le remplacement de la "strand notation" par la combinaison de Link et Pair qui ne détruisent pas les propriétés établies en cas de substitution.

Par exemple la séquence :

```
V←C;̃D
W←A;̃B;̃V
```

est équivalente à

```
W←A;̃B;̃C;̃D
```

En revanche la séquence :

```
V←C D
W←A B V
```

ne l'est pas.

Cet argument n'est point valable. En effet, il est du même ordre que le suivant :

Puisque 3 est équivalent à 2+1, alors 1 2 3 doit être équivalent à 1 2 2+1

Par ailleurs, les deux fonctions Link et Pair ne sont pas plus pratiques d'usage que la "strand notation".

Les faits parlent d'eux-même :

```
A B C D E
```

d'un côté.

```
A;̃B;̃C;̃D⍮E
```

de l'autre, sans compter toute l'attention nécessaire pour utiliser la fonction PAIR

(;̄) au lieu de link (;̃) entre *D* et *E*.

Un autre argument d'Iverson est que la "strand notation" établit une hiérarchie d'exécution avec une précédence sur les opérateurs et les fonctions, ce qui réduit à néant l'avantage lié à la précédence des opérateurs qui admettent comme argument une variable, et qui auront désormais besoin de parenthèses.

Soit l'expression :

```
Z ← A +⍤B C
```

où A, B et C sont des tableaux. Iverson souhaite que l'opérateur ⍤ ait comme argument droit le tableau B. Or avec la "strand notation", ⍤ aura comme argument droit le vecteur `(B C)` ce qui provoque une erreur de syntaxe. Il faudrait écrire en APL2 :

```
          Z ← A  (+⍤B)  C
```

Deux réponses peuvent être faites à Iverson :

1. l'intérêt de la "strand notation" est bien plus grand que l'inconvénient de devoir utiliser des parenthèses pour grouper un opérateur avec ses arguments.
2. la "strand notation" n'empêche pas la prise en compte des souhaits d'Iverson. En effet, il suffit d'une légère modification de la syntaxe pour qu'un opérateur admette comme argument un terme simple, modification qui a d'ailleurs été réalisée dans la dernière version d'APL2. Ainsi :

   ```
   Z ← A +⍤ B C D
   ```

   est maintenant analysée comme :

   ```
   Z ← (A +⍤ B) C D
   ```

Nous adoptons la "strand notation" parmi nos extensions en soulignant trois points :

- la notation est compatible avec la notation des vecteurs de constantes d'APL

```
A B C  ↔  (⊂A),(⊂B),⊂C
1 2 3  ↔  (⊂1),(⊂2),⊂3
```

Afin que ce résultat soit identique à

```
1, 2, 3
```

il faudrait que l'enclose d'un scalaire reste un scalaire, ce qu'Iverson refuse.

- Aucune fonction ne peut à elle seule remplacer la strand notation. Autrement dit, il n'existe aucune fonction F telle que :

```
A B C ↔ A F B F C
```

- La "strand notation" n'est pas une fonction mais une notation. Elle a des implications sur la syntaxe.

Remarque: Nous avons utilisé la dénomination "strand notation" et non notation vectorielle car certaines propriétés de la notation vectorielle ne sont pas vérifiées par la "strand notation". Falkoff en cite trois [Falkoff81] :

1. **L'unicité :** la notation vectorielle est unique pour un vecteur donné. Elle est fondée sur la représentation des éléments numériques, or le vecteur 3 4 peut être également représenté avec la strand notation par (3) (4) , le vecteur (1 2) (3 4 5) par ((1 2)) (3 4 5).

2. **La sommation des longueurs :** si on juxtapose les représentations de deux vecteurs de constantes séparées par un espace, la longueur du vecteur de constantes ainsi représenté est la somme des longueurs des contributions.

   Or avec la "strand notation", 3 4 est un vecteur de longueur 2 et (7 8) est également un vecteur de longueur 2, mais leur juxtaposition 3 4 (7 8) est de longueur 3 et non 4.

3. **L'équivalence fonctionnelle :** dans la notation vectorielle on peut remplacer de manière arbitraire un espace (ou un blanc) quelconque de la représentation d'un vecteur de constantes par la fonction primitive de concaténation sans conséquence pour la valeur du vecteur.

   Ainsi :

```
1 2 3 ↔ 1,2,3
```

Or l'équivalent fonctionnel de la "strand notation" est d'enclore chaque élément et de les concaténer tous ensemble. Dès lors il faut être très attentif dans l'application de cette règle. En effet :

```
( 2 3) (4 5 6)
```

et

```
(2 3),⊂(4 5 6)
```

produisent des résultats différents.

En réalité, ces objections nous semblent due à une confusion sciemment entretenue par les objecteurs entre syntaxe et sémantique. La "strand notation" introduit un nouveau mécanisme syntaxique, qu'il est vain de tenter de relier à la sémantique d'autres mécanismes.

4.4 LES OPERATEURS

Les opérateurs sont un des concepts les plus importants du langage APL et offrent sans doute des possibilités d'extension très intéressantes. APL possède un jeu très restreint d'opérateurs qui de surcroit ne peuvent être utilisés qu'avec des fonctions primitives spécifiques.

Les propositions qui suivent ont pour but de lever ces limitations et de satisfaire les nouveaux besoins induits par les tableaux généralisés.

4.4.1 L'opérateur EACH

Il est représenté par le symbole ¨. C'est l'action terme à terme. Dans APL, une fonction primitive scalaire est une fonction qui s'applique élément par élément à ses opérandes. Ainsi :

```
× 2  0 ¯5 3 ↔ (×2),(×0),(×¯5),(×3)
            ↔ (1),(0),(¯1),(1)
            ↔ 1,0,¯1,1
2 3 4 + 5 1 ¯2 ↔ (2+5),(3+1),(4+¯2)
               ↔ (7),(4),(2)
               ↔ 7 4 2
```

Si F dénote une fonction primitive scalaire monadique, V un vecteur et I un indice scalaire quelconque de V, l'identité suivante est valide :

```
(F V)[I]↔ F V[I]          (1)
```

Les tableaux généralisés offrent un cadre dans lequel chaque fonction pourrait être source d'une **fonction scalaire associée**

La fonction scalaire associée dérive de la source par l'application d'un opérateur **each**

A partir de maintenant, nous représenterons les tableaux généralisés par leurs valeurs entre parenthèses. Pour bien comprendre l'action de EACH, prenons l'objet A :

```
              A
(2 7)  (1 2 3)  (1 2
                 3 4 )
```

A est un vecteur enclos de 3 éléments dont le premier est un vecteur à 2 éléments, le deuxième un vecteur à 3 éléments et le troisième est une matrice de taille 2 2 :

```
          ρA
3
```

ρ¨ est la fonction scalaire associée à ρ et fournit la dimension de chaque élément de chaque élément de son paramètre.

```
        ρ¨A
(2)  (3)  (2 2)
```

De même :

```
        2 3 ρ 4 5
4 5 4
5 4 5
        2 3 ρ¨ 4 5
(4 4)  (5 5 5)
```

De façon générale, si G est une fonction monadique, W un vecteur enclos dont les éléments sont des tableaux appartenant au domaine d'application de G et si I est un index scalaire quelconque de W, alors pour la fonction scalaire associée *G¨* on a l'identité suivante :

`(G¨ W)[I]↔ ⊂G⊃W[I]        (2)`

Si on adopte le point de vue qui consiste à dire que dans APL l'opérateur EACH a existé implicitement pour les extensions scalaires, alors la fonction scalaire associée d'une fonction primitive scalaire doit être identique à cette primitive.

L'identité suivante est valable par conséquent pour toute fonction primitive scalaire F :

`F ↔ F¨`

En particulier, si F est monadique, l'identité (2) devient (avec F pour G)

`(F V)[I]↔ ⊂F⊃V[I]        (3)`

En comparant les identités (1) et (3), on remarque la nécessité pour les deux primitives enclose et disclose de n'avoir aucun effet sur les scalaires simples.

`⊂3 ↔ 3`

et

`⊃3 ↔ 3`

Bien plus, l'identité (3) suggère que les fonctions primitives scalaires soient **pénétrantes** sur les tableaux généralisés.

Rappelons que c'est le choix préconisé par Jim Brown [Brown81].

4.4.2 Extensions des opérateurs d'APL

Les opérateurs primitifs tels que **la réduction, la propagation** (scan), **le produit interne** et le **produit externe** ne sont définis que pour les fonctions primitives scalaires diadiques.

Leur définition peut être étendue afin qu'ils admettent n'importe quelle fonction (primitive, définie, ou dérivée) et n'importe quel tableau comme argument.

L'approche d'APL2 se fonde sur le fait que dorénavant, toute fonction admet une fonction scalaire associée. Par conséquent, l'extension du domaine des opérateurs à une fonction quelconque se fait tout simplement en utilisant sa fonction scalaire associée.

Ainsi, soit F une fonction et A un tableau.

F/A est définie en utilisant implicitement la fonction ***F¨*** à la place de F.

Et comme :

F¨ ↔ ***F***

pour toute fonction scalaire, **le comportement actuel** des opérateurs ne change pas.

Remarques : Dans APL, un opérateur s'applique à une fonction pour fournir une nouvelle fonction comme résultat. Avec cette définition, il est impossible de dire si / est un opérateur comme dans :

```
      +/ 1 2 3
6
```

ou une fonction diadique comme dans :

```
      1 0 1 / 1 2 3
1 3
```

Dès lors que l'on étend le domaine des opérateurs aux fonctions et aux tableaux, l'ambiguité est levée et / devient un opérateur monadique dans les deux cas. Cette extension nécessite que les **opérateurs** aient une **valence fixe.**

Une autre difficulté de l'extension des opérateurs aux fonctions non scalaires est leur comportement face aux tableaux vides[4].

4.4.3 Les opérateurs définis

Il s'agit de donner la possibilité de définir ses propres opérateurs comme c'est le cas des fonctions définies. Cette extension ne pose pas de problème particulier dès lors que l'on suit la même philosophie que les fonctions utilisateurs.

---

[4] On peut se référer à [Brown81] pour une analyse du problème.

Ainsi les opérateurs définis sont aux opérateurs primitifs, ce que les fonctions définies sont aux fonctions primitives d'APL.

Dans APL2, on propose les formes suivantes pour l'en-tête des opérateurs définis.

Formes avec résultat explicite	Formes sans résultat explicite
`Z ← (LO MOP) R`	`(LO MOP) R`
`Z ← L (LO MOP) R`	`L (LO MOP) R`
`Z ← (LO DOP RO) R`	`(LO DOP RO) R`
`Z ← L (LO DOP RO) R`	`L (LO DOP RO) R`

où Z est le nom du résultat, L l'argument gauche, R l'argument droit, LO l'opérande gauche, RO l'opérande droit, MOP nom d'opérateur monadique, DOP nom d'opérateur diadique.

Comme montré dans [Girardot85], on ne décrit pas en fait le comportement de l'opérateur, mais celui de la fonction définie qu'il produit.

5 LES OPERATEURS DE COMPOSITION D'IVERSON

Nous analyserons ici, les extensions préconisées par IPSA qui nous semblent intéressantes et qui ne constituent qu'une partie des propositions qu'on trouvera dans [Iverson80].

Avant de présenter ces opérateurs, disons quelques mots de l'approche d'IPSA. Elle est fondée sur une réflexion en profondeur sur ce qui fait la **différence entre les fonctions scalaires et les fonctions mixtes** (la notion de rang) ainsi que sur le **mécanisme des fonctions scalaires** afin de l'étendre aux représentations scalaires de structures.

Il s'agit de définir des opérateurs qui multiplient les possibilités de l'APL classique tout en offrant un mécanisme systématique de manipulation de tableaux enclos, en les couplant avec les primitives ENCLOSE et DISCLOSE.

L'aspect essentiel de ces opérateurs de composition est de clarifier le comportement général des fonctions primitives, et de permettre d'exploiter la notion de **rang d'une fonction.**

La notion de rang d'une fonction apparaît pour la première fois dans [Iverson78]. Le mot rang, ici, est pris au sens APL, à savoir : un scalaire est de rang 0, un vecteur de rang 1, une matrice de rang 2, etc ...

Une fonction se caractérise par le rang de ses arguments auquel elle s'applique **naturellement,** ainsi que par le rang du résultat qu'elle fournit.

Par exemple, IOTA monadique (⍳) a un "rang d'argument" 0 (il s'applique aux scalaires) et un "rang de résultat" 1 (il fournit des vecteurs). L'inversion de matrices (⌹) a un rang d'argument 2 et un rang de résultat 2. Les fonctions scalaires (+, −, ×) ont des rangs d'arguments 0 et des rangs de résultats 0. D'autres fonctions (la linéarisation par exemple) ont un rang d'argument indéfini.

L'extension d'une fonction à des arguments de rang plus élevé, se fait en appliquant la fonction "terme à terme", terme voulant dire **"la cellule de la structure qui a le rang naturel de l'argument de la fonction".**

Si les fonctions scalaires s'étendent sans problème dans APL aux tableaux, c'est que les cellules auxquelles elles s'appliquent sont de rang 0 de même que le résultat qu'elles produisent.

Le miroir diadique (rotation) a des arguments de rang 0 (à gauche) et 1 (à droite) et un résultat de rang 1. Utilisé avec des matrices, il s'appliquera aux lignes ou aux colonnes, qui sont les cellules de rang 1 contenues dans ces matrices, l'argument gauche étant un vecteur.

Ainsi l'extension naturelle de **l'inversion de matrice** (le domino) à un tableau de dimension 3 4 4 serait d'appliquer l'inversion aux trois **cellules de rang 2,** en considérant le tableau comme un vecteur de trois matrices 4 4.

Cette notion de rang des arguments et des résultats d'une fonction, définissant la cellule sur laquelle s'appliquera la fonction confrontée à un argument de rang plus élevé, est pour l'instant, implicite en APL. C'est une notion plus ou moins attachée à chaque fonction sur **laquelle l'utilisateur n'a pas prise.**

Soit :

```
      V← 10 20 30
      M← 3 3 ⍴⍳9

      M                        V
1 2 3                 10 20 30
4 5 6
7 8 9
```

Il est tentant de faire *M+V* en espérant obtenir :

```
      11 22 33
      14 25 36
      17 28 29
```

Ce qui supposerait que **l'addition change de rang** et comprenne que la **cellule** n'est plus un scalaire, **mais un vecteur** et donc que M n'est pas un tableau de scalaires, **mais un vecteur de vecteur** et donc, etc ...

Par ailleurs un tableau de dimensions 3 5 2 2, est-il une matrice 3 5 de matrices 2 2, ou un vecteur de 3 tableaux de dimensions 5 2 2 ?

La première vertu des opérateurs de composition est de **laisser l'utilisateur maître du rang des fonctions,** ce qui se fait en composant la fonction avec un vecteur de dimensions.

Nous ne nous étendrons pas sur les techniques de cet usage des nouveaux opérateurs dont un argument est une fonction, l'autre un vecteur ; beaucoup d'exemples sont cités dans [Iverson80]. Mais nous retiendrons que ce nouveau degré de liberté permet d'imposer à toute fonction (primitive ou dérivée) de travailler sur **des cellules définies** dans une structure qui en constitue le **cadre.**

En revanche, nous allons analyser ces opérateurs de composition lorsque leurs deux arguments sont des fonctions.

Iverson préconise trois opérateurs de composition qui sont :

⍤	*ON*
⍥	*UPON*
¨	*WITH*

Nous utiliserons le signe

→←

pour indiquer les identités (ou équivalences) entre les **cellules** des arguments par opposition à

←→

dont l'usage est d'indiquer l'identité globale.

Soient F et G deux fonctions, et α et ω des tableaux. Les fonctions **enclose** et **disclose** se notent respectivement < et >.

5.1 L'OPERATEUR ON

C'est le signe ⍤ . Sa définition est la suivante :

monadique :

F ⍤ *G* ω →← *F* *G* ω

diadique :

α *F*⍤*G* ω →← (*G* α)*F*(*G* ω)

Il représente l'application de la fonction F sur le résultat de la fonction G, et correspond donc à la composition des fonctions au sens mathématique.

Pour bien comprendre la différence entre l'identité globale et l'identité entre cellules, montrons la différence entre :

⍉⌹A

et

⍉⍥⌹A

où A est de dimension 3 4 4.

Le domino (⌹) est une fonction de rang 2, qui s'applique à chaque cellule (couche) de A.

Dans le cas de

⍉⌹A

on applique d'abord le domino aux cellules de A obtenant un tableau de dimension 3 4 4 dont chaque couche est l'inverse de la couche correspondante de A. Ensuite on transpose le **tout,** obtenant un résultat de dimensions 4 4 3.

Dans le cas de

⍉⍥⌹A

c'est la transposition **composée** avec le domino que l'on applique à chaque cellule.

⍉⍥⌹A →← ⍉⌹A

Le résultat final reste de dimension 3 4 4 comme A, mais **chaque couche** est la **transposée de l'inverse** des couches correspondantes de A.

Donnons quelques exemples d'application de ON, en utilisant ENCLOSE et DISCLOSE de chez IPSA.

Soit :

```
      VILLE ← ('BAMAKO'),('SEGOU '),('NIONO ')

      >VILLE
BAMAKO
SEGOU
NIONO

      ρ>VILLE
3 6
```

Appliquons la **composition** de rho et disclose

```
      ρ⍤>VILLE
6
6
6
```

Cependant, disclose ne s'applique pas si les mots n'ont pas la même longueur :

```
      >('BAMAKO'),('SEGOU'),('NIONO')
```

donnerait une erreur chez ISPA.

Pour s'en sortir, il suffit de composer disclose avec la fonction **prend,** et d'utiliser comme argument gauche le scalaire 6 représentant la longueur du plus grand mot.

```
      6↑⍤>('BAMAKO'),('SEGOU'),('NIONO')
BAMAKO
SEGOU
NIONO
```

De manière plus générale :

```
      (⌈/,ρ⍤>VILLE) ↑⍤>VILLE
BAMAKO
SEGOU
NIONO
```

et

```
      ('N'=,1↑⍤>VILLE)/VILLE
```

donne les mots de VILLE commençant par ***N*.**

5.2 L'OPERATEUR UPON

C'est le symbole ⍥. Sa définition est la suivante :

monadique :

```
                    F⍥G ω →← F G ω
```

diadique :

```
                    α F⍥G ω →← F α G ω
```

Dans le cas monadique UPON est équivalent à ON.

En diadique, UPON est l'application de la fonction monadique F au résultat de la fonction diadique G. C'est une forme altérée de composition de fonctions.

F⍤G et *F⍥G* se différencient l'un de l'autre (malgré la ressemblance des symboles ⍤ et ⍥ par la valence utilisée pour la fonction G qui est diadique dans le cas du symbole le plus grand et monadique dans l'autre.

Donnons quelques exemples :

```
      A
SE
GA

      B
SA
KO


      A,⍤,B →← (,A),(,B)
SEGASAKO

      A,⍥,B →← ,A,B
SESAGAKO

      C<⍤÷ D ← ⌽ C ← 1 2 3 4
0 0 1 1

      C<⍥÷ D
(0.25) (0.66666667) (1.5) (4)
```

Dans le premier cas, la fonction primitive < a été comprise comme la fonction de relation "strictement inférieur à", donc diadique, et dans le second cas comme la fonction ENCLOSE, ce qui fait qu'on obtient un vecteur de scalaires enclos, car la fonction ENCLOSE n'est pas permissive sur les scalaires chez IPSA.

5.3 L'OPERATEUR WITH

C'est le symbole ¨ Sa définition est la suivante :

monadique :

F¨*G* ω →← *IG F G* ω

diadique :

α *F*¨*G* ω →← *IG* (*G* α) *F G* ω

La fonction dérivée *F*¨*G* est appelée la fonction duale de *F* par rapport à *G*.

IG indique la fonction inverse de *G* (au sens où le logarithme est l'inverse de l'exponentielle, l'enclose est l'inverse du disclose, la division monadique ou le transpose leurs propres inverses).

F¨*G* diffère de *F*⍤*G* uniquement par l'application supplémentaire de la fonction inverse de *G* à chaque résultat cellulaire.

La dualité des fonctions fut l'une des découvertes les plus profondes d'Iverson. Elle permet la **généralisation** de la **loi dite de Morgan.**

On sait que :

α∧ω ↔ ~(~α)∨(~ω)

la fonction NOT (~) qui intervient en tête du second membre, masque son rôle par le fait qu'elle est son propre inverse. Mais si on considère PLUS et MULTIPLIER, on a :

α×ω ↔ ★(●α)+(●ω)

où l'exponentielle (★) est l'inverse du logarithme (●).

Moins étant son propre inverse, son rôle est masqué dans l'identité suivante :

α⌈ω ↔ −(−α)⌊(−ω)

De manière plus générale, si H, F et G sont trois fonctions, et IG l'inverse de G, la dualité entre H et F par rapport à G, n'est pas (De Morgan) :

```
α H ω ←→ G(G α)F(G ω)
```

mais (Iverson) :

```
α H ω ←→ IG(G α)F(G ω)
```

Ainsi :

```
V ← 1 2 3 4 5
+\¨⌽V    →← ⌽+\⌽V
```

Car

⌽

est son propre inverse, et le résulat est :

```
15  14  12  9  5
```

De même :

```
          1 2 3 +¨÷ 2 3 4
0.66666667   1.2   1.71428571
```

car ÷ est son propre inverse.

Si on se reporte au paragraphe précédent, on peut conclure qu'une fonction avec l'opérateur EACH d'APL2 est sa duale par rapport à DISCLOSE.

```
α F¨ ω →← <(> α)F(> ω)
  F¨ ω →← <(F> ω)
```

le trema de F¨ indique ici le EACH de APL2, qui apparaît donc comme un cas particulier du WITH d'Iverson.

5.4 REMARQUE

Une fois qu'on a saisi que :

- la différence fondamentale entre les fonctions mixtes et les fonctions scalaires c'est le rang des arguments et du résulat ;

- les tableaux de tableaux sont manipulables grâce à la représentation scalaire (enclose et disclose),

alors les trois opérateurs de composition (ON, UPON, WITH) permettent à une **fonction arbitraire** F :

- de travailler sur des arguments de rang 0 :

```
F̶ö>
```

- de donner des résultats de rang 0 :

```
<öF
```

- de travailler comme des fonctions scalaires, (argument et résultat de rang 0) :

```
F¨>
```

ce qui offre un mécanisme général systématique, qui apparait clairement et qui exclut donc les encloses implicites comme dans APL2.

Par exemple dans APL2 :

```
     2 4 ?¨ 3 6
(2 1) (5 3 5 4 )
```

ici l'enclose des résultats individuels est implicite. Chez IPSA, pour obtenir le même résultat, il suffit d'écrire :

```
     2 4 <ö? 3 6
(2 1) (5 3 5 4 )
```

l'enclose apparaît clairement dans l'expression.

6 INTEGRATION DES EXTENSIONS

Au terme de cette analyse, nous nous apercevons qu'il n'est pas possible de faire une synthèse entre les propositions, tant elles émanent, dans certains cas, d'approches très différentes.

Même si les systèmes IPSA et APL2 ont des outils comparables, ils représentent deux conceptions différentes des tableaux généralisés en APL.

L'impression que nous donne le système APL2, est de faire éclater les contraintes liées aux tableaux homogènes de scalaires, ainsi que son adéquation (élargissement maximum des concepts APL) aux conséquences, et à l'usage des tableaux généralisés. Les fonctions primitives scalaires en sont les premières bénéficiaires.

Le système APL2 semble basé sur la volonté d'étendre les fonctions primitives et les opérateurs actuels, chaque fois que possible, au domaine des tableaux généralisés. En effet :

1. L'extension de la réduction, de la propagation, des produits intérieurs et extérieurs utilisant implicitement l'opérateur EACH.

2. La "strand notation" étend la représentation des vecteurs numériques.

3. L'extension de l'indexation utilise les tableaux généralisés.

4. L'extension de l'application des fonctions aux collections de sous tableaux se fait à l'aide de l'opérateur EACH, et en encapsulant seulement selon des axes spécifiés.

Rappelons que cette approche nécessite que **enclose**l' d'un scalaire soit un scalaire sinon les extensions 1 et 2 ne marchent pas.

Dans la mesure où beaucoup d'opérateurs et des primitives ainsi étendues pourraient aussi être effectivement étendus sans faire appel aux tableaux généralisés, on peut dire que cette approche est fondée sur la croyance que les tableaux généralisés sont potentiellement plus efficaces, pour la plupart des applications, que les tableaux simples.

Le système IPSA semble au contraire être fondé sur l'idée qu'il existe des alternatives aux extensions aux tableaux généralisés qui sont potentiellement puissantes sans les tableaux généralisés (opérateurs d'axes et de composition), mais qui peuvent aussi s'appliquer aux tableaux généralisés au moyen de l'usage explicite d'ENCLOSE et de DISCLOSE.

Ainsi la croyance sous jacente de l'approche d'IPSA semble être que la généralité est encore un des principes fondamentaux de la conception et du développement d'APL.

Il est donc clair qu'à partir de maintenant, la communauté APL doit se faire à l'idée qu'il va exister au moins deux familles de dialectes d'APL : ceux compatibles APL2 et ceux compatibles IPSA. C'est une situation connue déjà pour d'autres langages (LISP, PASCAL ...).

Pour l'intégration des extensions, des études ont été menés sur la syntaxe et la méthode d'analyse du langage.

Des travaux ont ainsi été réalisés par Girardot et Rollin (dans le cadre du projet APL 90), sur l'influence des extensions sur la syntaxe du langage, permettant une certaine formalisation de la syntaxe d'APL, et différents modèle de grammaires formelles ont été proposés [Girardot86, Rollin83]. La concrétisation de ces travaux fut la réalisation d'un générateur (écrit en APL), qui à partir d'une grammaire fournit l'automate qui permet de l'analyser.

Rappellons certaines conclusions de ces travaux :

- les extensions qui n'ont pas de conséquence sur la syntaxe :
    - l'ambivalence de toutes les fonctions primitives, c'était déjà le cas dans APL
    - l'ambivalence des fonctions définies
    - l'ajout de nouvelles fonctions primitives et opérateurs, l'analyse lexicale permettra de les reconnaittre
- la définition des opérateurs d'Iverson ne permettent pas de respecter toujours l'ordre d'évaluation définie par Iverson lui même
- Pour rester compatible avec la notation des vecteurs numériques, l'application de la "strand notation" doit être prioritaire sur toutes les opérations d'une phrase APL
- la proposition d'Iverson de remplacer les crochets d'axe et d'indexation (qui s'intègrent mal aux trois entités du langage : tableau, fonction ou opérateur) par deux opérateurs nouveaux permet de supprimer une anomalie du langage, mais elle génère en même temps une incompatibilité avec APL.

En ce qui nous concerne nous avons opté dans un premier temps pour la compatibilité avec APL2 d'IBM, notre architecture de machine nous permettant théoriquement d'exhiber facilement un modèle compatible IPSA. Par ailleurs les propositions d'APL2 sont celles qui respectent le plus nos critères (compatibilité, simplicité ...).

Nous verrons enfin que l'adoption du modèle APL2 va avoir des conséquences importantes dans la seconde partie consacrée à l'extension orientée objet.

CONCLUSION PREMIERE PARTIE

La description complète de la réalisation a fait l'objet de nombreux documents techniques [Girardot-Mireaux-Sako85], et sa présentation n'apporterait que peu d'intérêt dans cette thèse. C'est pourquoi pour finir cette partie, nous nous contenterons de rappeller quelques originalités et caractéristiques du système APL 90 :

- Il gère une mémoire virtuelle paginée, qui est matérialisée par un fichier standard UNIX ou par un disque physique complet. La taille de cette mémoire virtuelle peut être quelconque (jusqu'à 2**53 octets), et n'est donc limitée en pratique que par l'espace physique disponible en mémoire secondaire.

- APL 90 sait gérer des données très grande taille : 2**31 éléments. Les données APL 90 sont des variables sans dimensions, ou des tableaux à un ou plusieurs indices. Le rang d'un tableau (nombre d'indices) peut atteindre 255.

- APL 90 optimise la représentation des objets en compactant les données ou en utilisant des représentations spéciales pour certaines entités (scalaires simples, vecteurs courts, progressions arithmétiques), afin d'économiser la mémoire et de diminuer les temps de traitement.

- Il permet le partage de valeurs entre différents objets d'une même zone de travail ou de différentes zones de travail d'une même mémoire virtuelle.

- Le nombre de symboles que sait gérer le système n'a qu'une limitation théorique : 2**31. Les tables de symboles sont organisées sous forme d'arbres HBB (Height Binary Balanced Tree), ce qui assure d'excellentes performances. La taille de ces tables augmente automatiquement en fonction des besoins de l'utilisateur.

- APL 90 travaille avec un jeu de caractères étendu, recouvrant tous les caractères ASCII. En particulier, ceux-ci sont utilisables à l'intérieur des chaînes et des commentaires, et les identificateurs peuvent comporter des lettres minuscules, ce qui permet d'utiliser pour les applications des terminaux "normaux".

- Il offre une compatibilité avec le système APL 2 d'IBM.

conclusion partie 1

DEUXIEME PARTIE : UNE EXTENSION D'APL VERS LES OBJETS

" What is object oriented programming ? My guess is that object oriented programming will be in 1980's what structured programming was in 1970's. Everyone will be in favor of it. Every manufacturer will promote his product as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is. "

Tim Rentsch

Dans la communauté informatique, il est couramment admis plusieurs paradigmes de programmation que nous classerons en trois catégories : la programmation orientés vers les **procédures,** les **objets,** les **règles.**

i. La programmation orientée vers les procédures

C'est le modèle dominant dans l'informatique d'aujourd'hui. Tous les langages classiques (PASCAL, LISP, APL ...) en relèvent. Deux sortes d'entités sont considérées : les **procédures** et les **données.**

Les premières sont **actives,** les secondes **passives.**

Les programmes sont organisés sous forme de procédures, des effets de bords se produisant lorsque 2 procédures partagent la même donnée et la modifient séparément. L'adoption de la programmation structurée est censée limiter ces effets de bords. Ce paradigme de programmation est le plus connu, le plus classique, le plus ancien et le plus éprouvé.

ii. La programmation orientée vers les objets

La programmation orientée vers les objets est née principalement des langages SIMULA et Smalltalk. Dans ce style de programmation, données et procédures ne sont pas séparées mais forment conjointement des entités appelées **objets.**

Les objets ont des procédures locales (les **comportements**) et des données locales (les **attributs**). Toutes les actions de ces langages peuvent se décrire comme la passation de **messages** entre objets, chaque objet produisant sa propre interprétation du message reçu.

Un aspect important de ces langages est l'existence d'un graphe **d'héritage** qui permet de structurer hiérarchiquement les objets en **classes** et sous-classes, chaque objet pouvant alors hériter des propriétés (comportements et attributs) des classes auxquelles il appartient.

iii. **La programmation orientée vers les règles**

Dans la programmation par règles, le comportement du système est dicté par des ensembles de couples **conditions-actions** ou **prémisses-conclusions.**

Dans chacun d'eux l'invocation d'une règle est guidée par **filtrage** ("pattern matching") sur les données.

Le langage PROLOG est évidemment le représentant le plus notoire de ce style de programmation.

Les deux derniers paradigmes de programmation sont plus récents et moins explorés. On rencontre également les vocables de programmation impérative, programmation fonctionnelle, programmation logique et programmation dirigée vers les données. A part la programmation logique qui se fonde sur une théorie permettant de représenter des énoncés et de de vérifier de manière formelle leur validité, les autres modèles nous semblent être plus des styles de programmation (on peut exhiber des exemples de programmation fonctionnelle dans des langages procéduraux comme LISP ou APL) ou des approches plus ou moins confuses des trois types déja cités, que de véritables paradigmes de programmation. C'est notamment le cas de la programmation orientée vers les données qui établit, entre données et procédures, un mode d'invocation original : des **actions-réflexes** sont déclenchées par un accès spécifique (lecture, écriture ou autre) à une donnée. Il s'agit, littéralement, de "réflexes".

La programmation orientée vers les données[1] est représentée essentiellement par les **frames** [Minsky75] et les **valeurs actives des flavors** [Moon81]. Ce style nous semble relever autant de la programmation objet que de la programmation procédurale.

Les langages de programmation **"orientés objets"** (L.O.O) connaissent aujourd'hui une vogue certaine dans le monde de l'informatique. C'est ainsi qu'il existe de nos jours plusieurs dialectes de LISP intégrant le type "objet" (FLAVORS [Moon80], ORBIT [Steele75], GLISP [Novak82], LOOPS [Bobrow82]), ainsi que divers langages objets directement inspirés de Smalltalk,

---

[1] **Il ne faut pas confondre la programmation orientée vers les données avec la programmation dirigée par les données ("data-driven programming"). Dans la programmation dirigée par les données, des attachements procéduraux**

permettent de définir des propriétés fonctionnelles pour des symboles.
sans parler des multiples extensions objet greffées sur Pascal, C, et autres Forth.

Est-ce un phénomène de mode ou la réponse à un réel besoin ?

Comme le fait remarquer Cointe [Cointe84] deux explications prédominent.

La première est d'ordre pratique : le développement d'environnements de programmation des machines LISP et Smalltalk (voire des machines pour la bureautique comme le système STAR) nécessite une gestion sophistiquée d'un écran graphique par un système de multifenêtrage. Or la réalisation d'un tel système semble largement facilitée par les concepts d'objets, de classe et d'héritage.

La seconde est certainement liée à l'essence même de la programmation en Intelligence Artificielle. Celle-ci travaille sur le raisonnement et fait moins appel à l'optimisation et à l'étude théorique d'algorithmes déjà connus, qu'à une programmation incrémentale qui tente d'expliquer des phénomènes aussi complexes que la reconnaissance de la parole ou la synthèse de la voix. Il s'avère que les langages utilisés dans ce dernier cas doivent satisfaire au moins trois critères :

1. **la modularité** : elle permet de décrire une situation en termes de sous-situations élémentaires. La modification d'un niveau d'algorithme ne doit nécessiter ni la connaissance, ni la retouche de ses voisins.

2. **l'extensibilité** : les phénomènes à modéliser font le plus souvent appel à des structures de contrôles non disponibles. Il faut donc pouvoir étendre facilement le langage hôte. Ainsi les structures de contrôle les plus souvent disponibles (récursives et itératives) s'avèrent mal adaptées à la manipulation efficace d'une base de données, à la mise en évidence de sous-structures (description abstraite d'arbre binaire, de programme LISP), ou encore au développement de mécanismes complexes comme les processus, les interruptions et les échappements.

3. **la multiplicité des représentations** : la compréhension d'un objet complexe, s'appuie sur différentes représentations ou plus exactement sur différentes visions de cet objet.

Il se trouve que les concepts d'objets, d'acteurs et de transmission de messages introduits dans les années 1970 par C. Hewith et A. Kay satisfont ces trois exigences.

Il existe aujourd'hui des langages procéduraux non interactifs comme PASCAL et C qui intègrent certains aspects de la programmation orientée objet (P.O.O). Chacun de ces langages développe à sa manière les techniques de la

P.O.O.

APL ne saurait rester à l'écart de ce grand mouvement. C'est pourquoi, nous allons dans les pages suivantes analyser les concepts sur lesquels reposent la programmation orientée objet, ce qu'ils apportent de nouveau. Ensuite, nous préciserons les motivations de l'extension d'APL aux concepts d'objets, nous exposerons les problèmes posés par la P.O.O dans un univers APL et nous proposerons des choix pour une extension d'APL 90 orientée objet. Enfin, nous expliciterons les structures de données internes utilisées pour la réalisation.

CHAPITRE 3

Chapitre 3

LA PROGRAMMATION ORIENTEE OBJET

1 LES CONCEPTS

Dans la programmation orientée objet, les concepts sont peu nombreux mais s'appliquent uniformément. Il s'agit respectivement de **l'objet,** de la **transmission de message, de classe,** de **l'instanciation,** et enfin, **d'héritage.**

L'universalité de ces concepts a permis d'aborder avec succès différents domaines de l'informatique, allant de la spécification de type abstrait (CLU), à celle de logiciel pédagogique (LOGO et BOXER) en passant par les systèmes experts, la représentation de connaissances (L.O.OPS et MERING) et l'informatique musicale (FORMES).

L'approche objet repose sur le souci de réunir l'information manipulée par un langage avec les programmes qui traitent cette information. Il s'agit **d'encapsuler** les données et les procédures qui manipulent ces données. C'est le premier principe de la programmation orientée objet.

1.1 L'OBJET

Un objet est défini comme la réunion d'une base de données et d'une base de procédures, la première caractérisant les connaissances (ou champs) de l'objet, la seconde ses actions potentielles (ou méthodes). Un objet est donc vu comme une entité autonome, autodocumentée et fournissant son interface de communication avec le monde extérieur.

Le terme objet a connu des formulations très variées. Dans [Minsky75] la notion de "Frames" ou schémas servait à décrire la compréhension des images et du langage naturel. Les informations sont représentées à l'aide de schémas, c'est à dire d'entités de forme prédéfinie dont il suffit de remplir les cases laissées libres. Un schéma est formellement une liste de couples <attribut, facette>, où une facette correspond à un élément de description d'un attribut.

Dans SIMULA [Dahl70,Birtwistle73], on vit apparaître les mots "formes" et "classes". Une nouvelle vision de la programmation était offerte en introduisant des concepts de structuration par classes et représentants (instances) et un mécanisme de contrôle généralisé par envoi de messages. Chaque élément du langage (objet) est décrit par l'ensemble de ces champs locaux et par la liste de ces méthodes : comportements qui sont évalués à la réception d'un message.

Aujourd'hui, c'est le terme objet qui (grâce peut être à sa généralité ou à son imprécision) apparaît dans tous les langages cités plus haut. Citons quelques exemples d'objets informatiques tirés de [Cointe86].

1.1.1 La tortue LOGO

La tortue LOGO qui bien que LOGO ne soit nullement un L.O.O, apparaît comme une entité détenant dans sa base de données un cap, un couple de coordonnées, la position de la plume, sa couleur, sa fréquence de reception, et capable d'exécuter un ensemble d'actions : lever la plume, descendre la plume, avancer, reculer, tourner à droite et à gauche.

Pour fixer les idées, le micro-monde de la tortue peut être représenté informellement par :

```
   ETAT = [(nom natacha)
           (cap midi)
           (x 10)
           (y 40)
           (couleur rouge)
           (fréquence 200 Hz)
           (position plume levee)
          ]

ACTIONS = [(lp (self) instruction pour baisser
           (dp (self) instruction pour descendre
                      la plume)
           (av (self nb-pas) corps de methode...)
           (re (self nb-pas) ...)
           (dr (self nb-degres) ...)
           (go (self nb-degres) ...)
          ]
```

1.1.2 La note de musique

La note de musique dans le langage FORMES est le regroupement de données (les attributs statiques) constitué par un nom, un temps de début, une durée, une hauteur, une amplitude, un timbre, un octave, et de méthodes (attributs dynamiques) gérant les procédures d'activation : jouer pour synthétiser la note, transposer pour changer son octave, ralentir pour raccourcir la durée, afficher pour insérer la note dans la partition vidéo et éditer pour imprimer le code associé.

Une représentation de la note serait :

```
      ETAT = [(nom DO)
              (temps-debut 0.2 s)
              (duree blanche)
              (hauteur 130 Hz)
              (octave 3)
              (timbre violon)
              (amplitude 80 Db)
             ]

   ACTIONS = [(jouer (self synthe) ... )
              (ralentir (self n-duree) ... )
              (transposer (self nb-octave) ...)
              (afficher (self partition) ...)
              (editer (self) ...)
             ]
```

1.1.3 Objet et type abstrait

La notion de type abstrait [Liskov84] s'apparente à celle d'objet si l'on accepte d'associer à chaque champ un domaine de définition. Ce choix n'est pas celui des langages objets issus de LISP ou Smalltalk mais correspond à la définition du langage CLU par B. Liskov.

Ainsi un nombre complexe sera vu comme un objet caractérisé par sa partie réelle, sa partie imaginaire, son angle polaire et son rayon vecteur et répondant à un ensemble des fonctionnalités : des lois de composition internes (+ * - conjugué) ou externe (conversion de type : complexe en tant que réel) par exemple. De même un rationnel sera défini par son numérateur, son dénominateur et les fonctionnalités associées.

Toujours dans l'approche "type abstrait", le concept d'objet permet de représenter des structures de données classiques comme les "collections", les "ensembles", les "queues" et les "piles".

Par exemple, une pile sera définie comme un pointeur sur une structure ordonnée et un pointeur sur son dernier élément (sommet) et trois procédures d'ajout (empiler) ainsi qu'un prédicat (pilevide) renseignant sur l'état de la structure (vide ou pleine).

```
ETAT = [(sommet entier)
        (p tableau d'entiers)
       ]
ACTIONS = [(empiler (self x) ...0)
           (depiler (self) ...)
           (pilevide (self) ...)
          ]
```

1.1.4 Objet et représentation des connaissances : le robot SHRDLU

Le problème posé dans les années 70 par T. Winograd était celui de la représentation de l'univers d'un robot chargé de résoudre les problèmes de déplacements relatifs d'un ensemble de cubes. Pour les programmes SHRDLU et AZERTYOP [Greussay78, Wertz83], cet univers est représenté par une base de donnés globale regroupant les différentes relations (sur, sous et tenu) établies entre les différents cubes et le robot.

Pour décrire une scène à quatre cubes A, B, C et D dont les trois premiers sont empilés dans cet ordre, le quatrième étant tenu par le robot, on pourra utiliser cette liste d'items :

```
((A sur TABLE) (A sous B) (B sur A)(B sous C) (D tenu) (C sur B) (C
sous AIR))
```

On observera que le mouvement d'un cube nécessite l'interrogation de la base de données et éventuellement sa mise à jour.

Dans la démarche objet, un cube devient un système de représentation à part entière, il détient trois champs : un champ d'état indiquant si le cube est tenu ou non, et deux liens, l'un vers l'objet posé sur lui, l'autre vers l'objet sur lequel il repose.

Ce cube est en outre doté d'un ensemble de fonctions de manipulation permettant entre autres la mise à jour de ses champs, son affichage, ses déplacements etc .

```
ETAT = [(nom A)
        (dessus air)
        (dessous table).
        (etat pose)
       ] ACTIONS = [(initialises-toi (self) initialisation-standard-des-champs)
        (montres-toi (self) affichage-du-cube)
        (liberes-toi (self) dépile-les-cubes-posés-sur-lui)
        (poses-toi-sur (self un-cube) va-se-poser-sur-un-autre-cube)
        (tu-es-tenu (self) est-saisi-par-le-robot)
       ]
```

L'approche objet s'oppose donc à l'approche relationnelle (à la PROLOG) en décentralisant la base de données parmi les objets du système de représentation.

1.1.5 Objet et Frame (M. Minsky)

"Here is the essence of the theory : when one encounters a new situation (or make a substantial change in one's view of the present problem) one selects from memory a substantial structure called a frame. This is a remembered framework to be adapted to fit reality by changing details as necessary.

A frame is a data-structure for representing a stereotyped situation, like beeing in a certain kind of living room, or going to a child's birthday party. Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is about what to do if these expectations are not confirmed." [Minsky75].

La structure de "frame" (ou schéma) imaginée par Minsky pour l'étude de la psychologie cognitive a donné lieu à plusieurs réalisations informatiques dans le domaine de la représentation des connaissances.

L'exemple suivant est tirée de [Winston81], il caractérise le "frame" Henry comme un ensemble d'attributs ou "slots" (sex, height, etc ...) auxquels sont associées des "facettes" (value, if-needed, default etc ...) dénotant une valeur (éventuellement fonctionnelle) :

```
(Henry (sex(value(man)))
       (height(value(178)))
       (weight(if-needed(/ height 2)))
       (car(default(1)))
       (hobbies(value(jogging)(skiing)))
       (occupation(value(teaching)(research))) )
```

Un "frame" apparait comme un objet particulier sans actions sémantique mais dont l'ETAT est déterminé par un ensemble de champs dont les valeurs sont, soit obtenues directement ("value"), soit calculées par activation d'une fonction (démon lancé lors d'un accès explicite au champ : le démon "if-needed" peut déduire le poids de la taille si besoin est).

1.1.6 Objet et Structure de contrôle : les acteurs d'Hewitt

"An actor is a potentially active chunk of knowledge which communicates with other actors by sending polite messages. No member of a community of actors is allowed to treat any other member as an object, but must rather enter into a dialogue with it much as would happen in a non-oppressive human society. Each actors thus retains both independance and dignity." [Hewitt75a]

Pour un objet dont l'état se réduit à une seule action on est ramené au cas d'une fonction unique à laquelle est attaché un ensemble de données. Une telle fonction est appelée une fermeture, elle représente un acteur minimal associant à une valeur fonctionnelle un environnement clôturant les valeurs de ses variables libres. Un tel acteur permet de matérialiser l'état d'un calcul, donc un contexte d'exécution en particulier lorsque le programmeur utilise un style de programmation par "passage à la continuation" pour expliciter le déroulement des appels fonctionnels.

Ainsi si l'on considère la définition SCHEME [Abelson85] de la factorielle avec continuation :

```
(define(fact n c)
  ;(fact 5(lambda (x)x)) -> 120
  (cond
    ((eq? n 0) (c 1))
    (else(fact (-n 1)(lambda(r)(c(* n r))))))
```

On observera à chaque appel "récursif" de l'acteur fact la construction de l'acteur anonyme :

```
ETAT = [(n valeur-de-n-à-la-création-de-l'acteur)
        (c valeur-de-c-à-la-création-de-l'acteur)]

ACTIONS = [(lambda(r)(c(* n r)))]
```

obtenu comme évaluation de la lambda expression. Cet acteur mémorise dans son champ "c" la valeur courante de la continuation et dans son champ "n" la valeur courante du paramètre n.

1.1.7 Objet, fenêtres, menus et interfaces graphiques

En tant qu'objet, une fenêtre est définie par deux objets points représentant sa diagonale principale, l'aspect de son cadre, la texture de son fond, l'entête affichant son nom et l'icône la représentant désactivée. Les fonctionnalités associées règlent : son activation, sa fermeture, son affichage, ses déplacements, ses modifications en largeur ou hauteur, etc. La définition du menu détaillant le protocole d'utilisation d'une fenêtre est immédiat puisqu'il suffit d'afficher l'ensemble des noms de méthodes (sélecteurs) associés à l'objet correspondant. Pour ce qui concerne le système des fenêtres, il est représenté par un objet "queue de priorité" chargé de fixer l'ordre d'affichage sur l'ecran.

Ces nombreux exemples pris dans des domaines d'application très différents font apparaître l'objet comme un système autonome détenteur d'un ensemble de connaissances intrinsèques mais qu'il peut communiquer au monde extérieur. Reste à expliciter comment un tel objet peut transmettre ses connaissances en définissant le protocole utilisé, protocole connu sous le nom de Transmission de Messages.

1.2 LA CLASSE

Une classe est un couple <caractéristiques communes, comportements communs> qui décrit une entité abstraite. On parlera d'attributs (ou variables d'instances) et de méthodes de classe.

Une classe est donc une entité pluri-fonctionnelle dotée d'une base de connaissances : elle définit les objets et les types associés ainsi que les méthodes qui vont permettre de les manipuler.

Cette définition impose de considérer une classe comme un concept, qui peut se matérialiser par des représentants nommés instances. L'instance spécifie les caractéristiques locales de ce concept et suit les comportements globaux de celui-ci.

Une classe apparaît comme un moule car elle est avant tout un descripteur des potentialités qui seront associées aux instances.

Dès lors deux niveaux se dégagent dans le langage :

- celui de la manipulation des abstractions que sont les classes : création de représentants, héritages, modifications dynamiques de leurs propriétés, ajout de méthodes, etc.

- celui de la manipulation d'objets, les représentants des classes.

Quelle est la différence entre une classe et un objet ?

On peut y répondre en définissant la nature de la relation qui lie les classes entre elles d'une part, et celle qui lie une classe et les objets qu'elle engendre d'autre part. Dans le premier cas, la relation traduit les groupes d'entités satisfaisant la description, c'est le lien d'héritage. Deux classes sont reliées entre elles par une relation **d'inclusion.** Dans le second cas la relation traduit la structure des entités génériques, c'est le lien d'instanciation. Les instances sont reliées aux classes par une relation **d'appartenance.** Le propre d'une classe est alors d'être **génératrice d'une instance.**

1.3 INSTANCIATION

Le mécanisme de l'instanciation fixe le protocole de création dynamique d'un objet, il établit les relations sémantiques entre l'objet générateur et l'objet généré. Tout objet est instance d'une classe. **Une classe** est donc un objet particulier spécialisé dans la création d'autres objets par moulage.

L'idée est de regrouper les objets obéissant à la même sémantique c'est à dire partageant le même ensemble de fonctionnalités et le même jeu de champs mais se distinguant néanmoins les uns des autres par la valeur de ces champs.

Si nous reprenons l'exemple de la note de musique DO et si on définit un objet-note de nom LA :

```
      ETAT = [(nom DO)
              (temps-debut 0.2 s)
              (duree blanche)
              (hauteur 130 Hz)
              (octave 3)
              (timbre violon)
              (amplitude 80 Db)
             ]

   ACTIONS = [(jouer (self synthe) ... )
              (ralentir (self n-durée) ... )
              (transposer (self nb-octave) ...)
              (afficher (self partition) ...)
              (éditer (self) ...)
             ]
```

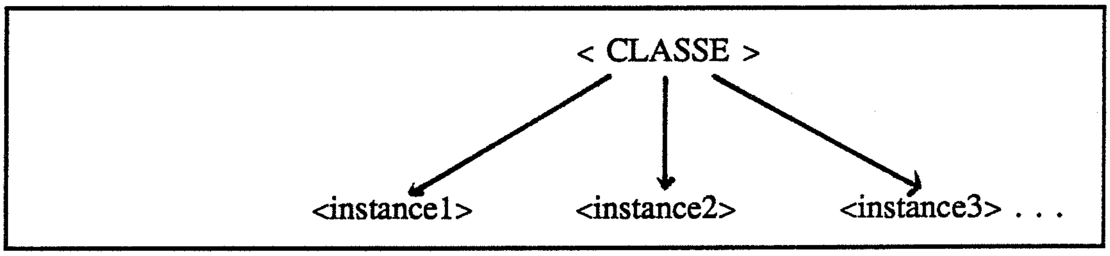
```
      ETAT = [(nom LA)
              (temps-debut 0.2 s)
              (duree blanche)
              (hauteur 440 Hz)
              (octave 3)
              (timbre violon)
              (amplitude 80 Db)
             ]

   ACTIONS = [(jouer (self synthe) ... )
              (ralentir (self n-durée) ... )
              (transposer (self nb-octave) ...)
              (afficher (self partition) ...)
              (éditer (self) ...)
             ]
```

On s'aperçoit aisément que les notes DO et LA partagent méthodes et champs (temps-début, durée, hauteur, octave, timbre, amplitude) mais se différencient par les valeurs de ceux-ci. On peut donc factoriser méthodes et champs en définissant une classe NOTE, le protocole d'instanciation ne se contentant que d'allouer une structure simplifiée qui pointe sur la classe (relation "isit" de Smalltalk) et les valeurs des champs.

Une représentation de la classe NOTE serait alors :

```
      NOM = NOTE
   CHAMPS = [(isit temps début durée hauteur
              octave timbre amplitude)]
 METHODES = [(jouer (self synthe) ...)
             (afficher (self partition) ...)
             (renverser (self) ...)
            ]
```

L'arbre d'instanciation devient :



L'instanciation pose essentiellement deux problèmes : la création d'un objet et l'initialisation des valeurs de ses attributs.

- **la création** : le principe consiste à générer systématiquement un objet à partir d'un objet modèle. Dans les L.O.O on utilise pour la création une transmission de messages de sélecteur particulier appelé généralement **NEW.** L'apparition physique de l'instance revient à allouer de la place mémoire à chaque attribut et à rendre les méthodes applicables sur l'instance ainsi créée.

- **l'initialisation** : il s'agit de résoudre un certain nombre de questions : peut-on spécifier des valeurs par défaut ? Peut-on initialiser les valeurs des attributs autrement que par défaut ? les valeurs doivent-elles rester indéfinies jusqu'à leur affectation explicite ? Dans les L.O.O, l'initialisation est traitée en général de manière dynamique. Les valeurs des attributs peuvent être spécifiées lors de l'instanciation même.

1.4 LA TRANSMISSION DE MESSAGES

L'une des idées fondamentales de la programmation orientée objet est de définir un protocole unique de communication : la transmission ou passation de messages. Elle permet d'activer un objet à la reception d'un message. L'envoi d'un message correspond à l'exécution d'une action générique.

L'envoi d'un message nécessite que soient connus :

- Le receveur : l'objet auquel on s'adresse
- L'action (sélecteur ou filtre) : l'ensemble des informations permettent à l'objet receveur de retrouver la méthode à appliquer.
- Les arguments éventuels : ils s'identifieront avec les paramètres formels de la méthode.

Dans la plupart des L.O.O la passation de message constitue une extension du classique appel de fonction. Ainsi, la forme la plus souvent utilisée est-elle :

(<receveur><action><argument 1><argument 2> ...)

dont la syntaxe souligne l'aspect fonctionnel du receveur.

A la réception d'un message, un objet doit exécuter l'action spécifiée par le message. Le s'lecteur du message est recherché parmi les méthodes de la classe de l'objet. En cas d'échec, une action par défaut est utilisée.

Dans un L.O.O, l'exécution d'un programme n'est qu'une succession de transmissions entre objets autonomes mais interconnectés.

1.5 L'HERITAGE

1.5.1 Présentation

L'héritage est la version moderne du mécanisme de préfixage des classes de SIMULA. Il s'agit de permettre à un objet d'hériter d'une ou de plusieurs classes un ensemble d'attributs statiques (les champs) et de comportements dynamiques (les méthodes), évitant du même coup toute répétition inutile de code et fournissant la programmation modulaire et structurée.

L'héritage est synonyme de partage d'environnements et de comportements. Il correspond à un ajout d'attributs et de fonctionnalités se traduisant pour une classe donnée par une union ensembliste entre ses propres champs (attributs ou méthodes) et ceux de la (des) classe(s) héritée(s). L'héritage peut être simple ou multiple. C'est un moyen de définir de nouveaux objets à partir d'objets existants

L'héritage implique une hiérarchie entre les classes. Toute classe définie comme sous classe d'une autre hérite automatiquement de ses champs et de ses méthodes.

Avant d'aborder les problèmes liés à l'héritage intéressons-nous aux résultats concernant les réseaux sémantiques, résultats qui traduisent la sémantique des deux liens d'instanciation et d'héritage [Brachman83].

Un réseau sémantique est constitué d'un ensemble de noeuds représentants des objets, des évènements ou plus généralement des concepts et d'arcs qui définissent les relations entre les noeuds. On peut également décrire un réseau sémantique par un ensemble d'entités <attributs, valeurs> constitué en graphe où les attributs sont des relations et les valeurs des pointeurs vers d'autres entités i.e. d'autres noeuds.

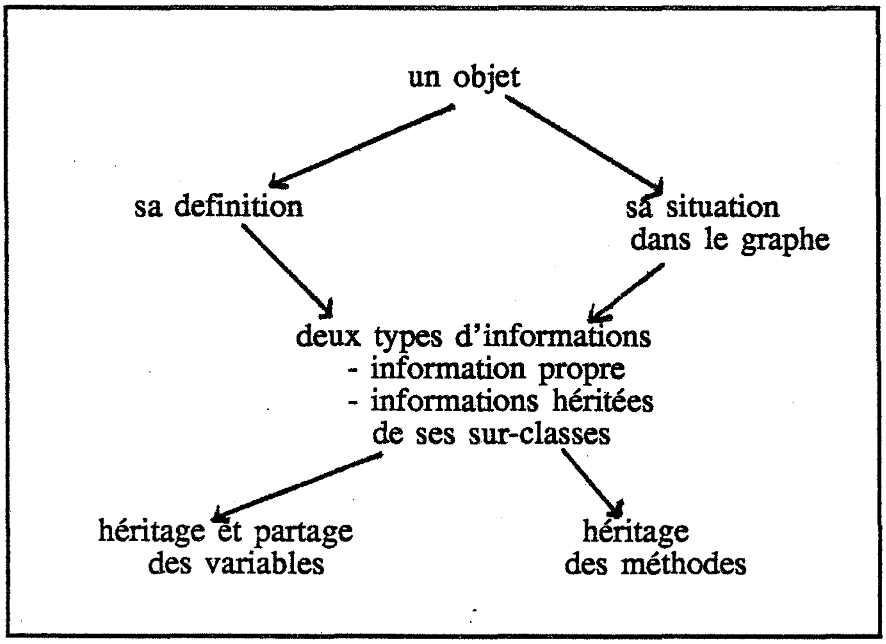
Le lien "est un" est souvent celui qui relie deux noeuds du gaphe. Ce lien renferme deux sens : un sens conceptuel et un sens ensembliste. Une classe (un noeud) est à la fois un concept (zèbre) et un ensemble (l'ensemble des zèbres). X "est un" Y signifie que X appartient à Y et que X a les propriétés de Y. X "est sous classe de Y" signifie que l'ensemble X est inclus dans Y, mais aussi que l'ensemble des propriétés de Y est inclus dans celui de X : X est par conséquent un **raffinement** de Y.

Le même lien "est un" peut-être alors un lien entre entités génériques, c'est à dire entre différentes classes, ou un lien entre une entité générique et un représentant, c'est à dire entre une classe et et une instance.

Dans le modèle objet, les entités génériques ou individuelles reliées par le lien "est un" sont des descriptions ou des concepts. La relation traduit soit la structure de ces entités, et c'est le lien d'instanciation, soit les classes d'objets satisfaisant la description et c'est le lien d'héritage.

L'héritage est donc un lien de modularité, il permet un deuxième degré de mise en commun des programmes en connectant entre eux des modules homogènes déjà structurés par le mécanisme d'instanciation.

Un objet dans un contexte d'héritage peut être caractérisé par le schéma ci-dessus tiré de [Benoit85] :



1.5.2 Les problèmes liés à l'héritage

De par la hiérarchie existant entre les classes, deux problèmes se posent :

- la représentation du graphe d'héritage
- la ou les stratégie(s) de parcours de ce graphe

Le graphe d'héritage représente une classification hiérarchisée par une relation "est un" que nous avons déjà évoqué. Les caractéristiques fondamentales de telles hiérarchies sont :

- L'héritage est une propriété logique attachée à la représentation de la hiérarchie entre les classes ;
- Les étiquettes associées aux noeuds d'une telle hiérarchie (vison des réseaux sémantiques) sont des prédicats unaires ;

- Des exceptions à l'héritage sont possibles : une sous classe peut masquer des comportements hérités.

1.5.2.1 Héritage simple

Dans le cas d'un héritage simple, la hiérarchisation se fait sous forme d'une arborescence dont la racine est une classe représentant l'ensemble des comportements communs à tous les objets.

Le problème de la recherche dans l'héritage simple est relativement facile à résoudre : il suffit de balayer l'arbre pour retrouver un comportement. Ce qui peut être pénalisant s'il faut remonter très haut dans l'arbre. Ce problème lié à l'implantation peut être résolu dans l'optique d'une compilation des appels de méthodes par rapport à une optique d'interprétation. La liste des méthodes à un noeud donné n'étant rien d'autre que celle des méthodes locales à laquelle on rajoute celle des méthodes héritées.

1.5.2.2 Héritage multiple

Si l'intérêt du concept d'héritage n'est plus à démontrer, l'expérience des L.O.O notamment de Smalltalk a fait apparaître rapidement que l'utilité du mécanisme d'héritage serait considérablement renforcée si une classe pouvait hériter de plusieurs super-classes. Notons que l'héritage multiple ne fait pas partie de la définition de Smalltalk-80, mais Borning et Ingalls [Borning82] ont défini les adjonctions nécessaires pour traiter ce mécanisme.

Le principe est simple :

- Une classe peut avoir un nombre quelconque de super-classes
- Un objet n'est cependant instance que d'une seule classe

Lorsqu'un objet reçoit un message, il y a d'abord recherche d'une méthode assocée dans le dictionnaire des méthodes de sa propre classe. En cas d'echec, la recherche se poursuit dans le dictionnaire des méthodes de ses superclasses immédiates, puis de leurs propres superclasses etc. Une erreur se produira si une même méthode est héritée de plusieurs superclasses immédiates d'une même classe.

Dans la version de Smalltalk-80 décrite dans [Goldberg83], il est possible d'accéder grâce à la pseudo-variable super à une méthode de la superclasse, même si le dictionnaire de la classe comporte lui-même une méthode de même sélecteur. Dans un contexte de multiples superclasses, il faudra donner plus de précisions, par exemple en utilisant des sélecteurs composés du type C.m où m est le sélecteur et C la classe à partir de laquelle doit se faire la recherche.

Une sous-classe hérite également des champs d'instance de ses superclasses. Le cas où plusieurs de ces champs portent le même nom est habituellement considéré comme une erreur. Le dispositif proposé par Borning et Ingalls se concrétise en particulier par la définition d'une sous-classe de la classe Metaclass (une métaclasse est une classe dont l'instance est elle-même une classe).

Dans l'héritage multiple, la hiérarchisation se fait sous forme d'un graphe qui peut être cyclique. Le choix d'une stratégie d'exploration du graphe à toujours été laissé à la responsabilité de l'implantation. D'où l'existence de différentes techniques telles que le parcours infixé (Flavors), la préfixation des méthodes (Mering). Dans MERING, il existe une priorité dans l'héritage. Un objet est toujours "plus" d'un certain type que d'un autre.

Il y a héritage des champs et des méthodes de toutes les superclasses avec des phénomènes de recouvrement en cas de conflits de noms de variables ou de méthodes . On peut admettre plusieurs stratégies de parcours du graphe d'héritage telles que :

a. Exploration en "profondeur", "premier rencontré"

b. Exploration en "profondeur ", "dernier rencontré"

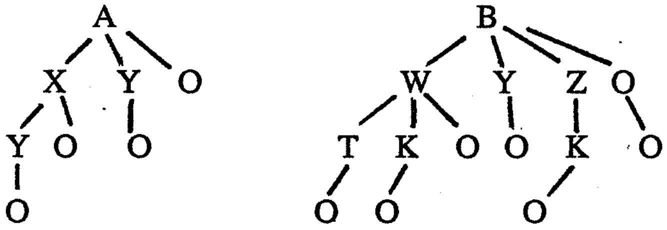
c. Explorer en "largeur", "premier rencontré"

d. Explorer en "largeur", "dernier rencontré"

Pour illustrer ces différentes stratégies, créons une classe C qui hérite des classes A et B avec :

H(A) = {X, Y, O}
H(B) = {W, Y, Z, O}
H(X) = {Y, O}
H(Y) = {O}
H(W) = {T, K, O}
H(Z) = {K, O}
H(K) = {O}
H(T) = {O}

H désigne l'héritage d'une classe, X, Y, Z, sont des classes, O étant la classe "Objet".



Alors l'héritage H(C) devient selon la stratégie :

a. H(C) = {A, X, Y, O, B, W, T, K, Z}
b) H(C) = {A, X, B, W, T, Y, Z, K, O}
c. H(C) = {A, B, X, Y, O, W, Z, T, K}
d. H(C) = {A, B, X, W, Z, Y, T, K, O}

O symbolise la classe "Objet" qui est la classe de butée. On remarquera que b) et d) ont l'intérêt de fournir la classe "Objet" en dernier. d) donne en outre l'ordre choisi (d'abord A, puis B, puis les autres, et O en dernier).

Dans une optique de compilation des méthodes, les problèmes de mise à jour deviennent très complexes.

Dans le concept d'héritage multiple, une question essentielle semble être : "quelle relation existe-t-il entre deux ascendants d'une même classe ?"

Autrement dit, quelle relation peut-on établir entre les 2 liens de cette classe et ses deux pères ?.

La majorité des L.O.O à héritage multiple considèrent que cette relation est une relation d'ordre. Si une classe A hérite de deux classes B et C, le lien "A est un B et C" est généralement interprété comme "A est un B puis C". Dans les héritages, cet ordre est appelé **multiplicité,** ou **priorité de la multiplicité.**

Notons que cette multiplicité se lit toujours de gauche à droite et trouve son interprétation précise dans l'algorithme de recherche.

La recherche dans la hiérarchie sert à résoudre les conflits issus de la présence de la même propriété dans plusieurs éléments de la hiérarchie d'un objet : de laquelle doit-il hériter ?

Les deux algorithmes de recherche dans le graphe d'héritage, les plus courants sont : en **profondeur** ou en **largeur** d'abord.

La justification de ces algorithmes réside en premier lieu dans le fait que les liens multiples sont présumés ordonnés, pour la relation de multiplicité. La recheche en profondeur d'abord est l'algorithme qui est adopté par la quasi totalité des L.O.O à héritage multiple. La recheche en largeur d'abord adoptée par quelques langages comme MERING, permet de reculer au maximum dans la recherche les objets les plus abstraits sous réserve que le graphe soit "equilibré", ce que ne fait pas la profondeur d'abord.

Dans ces deux méthodes, la recherche s'arrête (le long d'un chemin) lorsqu'est rencontré un noeud qui est déjà apparu : des occurences multiples, seule la première est conservée.

L'usage de ces algorithmes réside essentiellement dans leur simplicité.

Notons l'existence d'une méthode de recherche originale dans le langage YAFOOL [Ducournau86] qui est une tentative de réconciliation des deux points techniques précédents.

L'auteur justifie son algorithme et le caractère aberrant des deux précédemment cités par la phrase suivante :

"le graphe d'héritage forme un **un ordre partiel.** Or tout parcours de graphe, s'il s'arrête aux sommets déjà rencontrés produit un **ordre total** sur les sommets du graphe. Le moins que l'on puisse demander à cet ordre total, est d'être une **extension linéaire de l'ordre partiel** de la hiérarchie, **que parcourir un ordre le respecte."**

Or il est démontré [Habib86] que l'ordre obtenu par cet algorithme est une extension linéaire de l'ordre de l'héritage.

Donc dans YAFOOL la multiplicité n'est pas un ordre.

L'héritage multiple consiste en une union ensembliste des champs et des méthodes. Cette opération est statique. C'est à dire que l'héritage d'une classe est déclarée une fois à sa création.

2 PRINCIPE D'UNIFORMITE

Le principe d'uniformité se traduit par plusieurs aspects :

- Toute entité du langage doit être considérée comme un objet.

- Tous les objets communiquent entre eux en utilisant un protocole unique qualifié de transmission de message.

- Aucun objet n'a un statut particulier. Les objets définis par l'utilisateur sont traités comme les objets primitifs du langage. En d'autres termes, il n'y a pas d'objets de seconde catégorie.

3 LE PARALLELISME

Les concepts de base de la P.O.O commençant à se stabiliser, les problèmes liés au parallélisme gagnent de plus en plus d'intérêt dans la communauté des L.O.O.

Les objets sont souvent qualifiés d'entités rémanentes pour signifier que leur état (i.e. leurs attributs) se conservent d'une activation à l'autre. C'est pourquoi une classe peut être considérée comme définissant un univers protégé, encapsulé. L'univers construit est un monde stable que l'on peut quitter ; lorsqu'on y reviendra, tout sera dans le même état.

La rémanence a amené à aborder le problème de la signification d'algorithmes parallèles, l'interruption d'un programme puis sa reprise étant gérée automatiquement par le système [Serpette84].

De même, l'instanciation a conduit à envisager la question des protocoles de synchronisation entre différents objets simultanément actifs [Cointe84b].

Nous distinguerons cependant l'approche classique (Smalltalk), dans laquelle un objet n'est actif qu'à la reception d'un message, et jusqu'à transmission d'un résultat éventuel, un seul objet étant actif à la fois, de l'approche acteur, où les objets peuvent conserver une activité, qui se traduit par un parallélisme de l'exécution, avec tous les problèmes de synchronisation que l'on peut imager...

4 BREF HISTORIQUE DE LA P.O.O

SIMULA est l'ancêtre des L.O.O. Il est apparu en 1965 et unifia le couple <donnée, procédure> en un concept unique l'objet.

SIMULA est une extension d'ALGOL 60 incluant les notions de référence, de classe et de coroutine, ouvrant ainsi la voie à l'exécution parallèle.

Le mécanisme de préfixage des classes du langage a fait apparaître le concept d'héritage.

Le véritable démarrage des L.O.O eut lieu en 1969 lorsqu'A. KAY imagine de construire la machine FLEX. Il s'agissait de réaliser un ordinateur individuel doté de périphériques audio et vidéo, dont l'ensemble du système d'exploitation et du logiciel constituant l'environnement de programmation devait être réalisé à partir d'un langage unique ayant des concepts simples et efficaces.

Le choix du langage s'est opéré par référence à deux grands ancêtres : le langage LISP, à cause de la programmation interactive, fonctionnelle et symbolique, et le langage SIMULA pour son auto-spécification (c'est à dire que SIMULA est écrit en SIMULA). Le développement de ce langage a donné ce qu'on appelle aujourd'hui les machines Smalltalk.

Dès lors l'histoire de la programmation orientée objet suit d'assez près celle de Smalltalk. Le terme Smalltalk désigne comme APL un système et son unique langage. C'est un langage non déclaratif.

Smalltalk 72 a emprunté à SIMULA les concepts de classe et d'objet, en leur rajoutant la passation de messages. Cette notion de classe a largement dominé la conception du langage, même si l'influence de LISP est évidente dans la réalisation qui est fondée essentiellement sur la boucle système du type "(print(eval(read)))" qui en assure l'interactivité.

Smalltalk utilise une seule structure de donnée, **l'objet** et une seule structure de contrôle, la **transmission** de message. On voit l'analogie avec LISP qui utilise la liste comme unique structure de donnée, et l'application fonctionnelle comme unique structure de contrôle.

Smalltalk a fait, contrairement à SIMULA (trop dépendant d'ALGOL) un effort de consistance : toute entité manipulée par le langage doit être un objet.

Un objet Smalltalk est caractérisé par deux sortes d'attributs : les champs, ou instances de variables qui déterminent son état, les méthodes ou ensemble de procédures qui définissent ses actions potentielles.

Les objets Smalltalk sont activés par transmission de messages. L'instanciation d'un nouvel objet est obtenue en envoyant le message **new** à la classe génératrice.

Smalltalk 76 a généralisé le concept de préfixage des classes de SIMULA par une hiérarchisation des classes. Une classe peut être sous classe d'une autre classe ce qui permet l'héritage des champs et des méthodes. Le système apparait comme un arbre généalogique dont la racine est la classe OBJET.

Ensuite pour des besoins de consistance, les classes sont devenues elles-même des objets obtenus par instanciation de la classe CLASS.

Smalltalk 80 a introduit les notions d'héritage multiple, de méthode d'instance (propres à une instance d'un objet).

Le grand mérite de Smalltalk est d'avoir des concepts simples, ce qui a permis introduire aisément la de programmation orientée objet dans le monde des langages traditionnels. Smalltalk demeure de nos jours le modèle des langages et systèmes orientés objets.

5 QUELQUES LANGAGES ORIENTES OBJET

On rencontre de nos jours de nombreux langages de programmation orientés objet ou du moins pourvus d'une extension orientée objet. Il n'est pas possible de les citer tous. C'est pourquoi nous n'évoquerons que ceux qui nous semblent les plus représentatifs, hormis Smalltalk dont nous avons déjà parlé.

5.1 OBJECT PASCAL

Ce langage est une deuxième extension orientée objet du langage PASCAL. La première étant Clascal conçue uniquement pour le système LISA d'APPLE.

La syntaxe de Object Pascal a été conçue par l'équipe Clascal de chez APPLE en liaison avec Niklaus Wirth (le concepteur du langage PASCAL).

Le concept de classe est réalisé dans Object Pascal comme une extension de la structure RECORD du langage PASCAL.

En PASCAL, les records admettent uniquement des données comme composants des différents champs, alors que dans Object Pascal, les types d'objets (comme les classes) admettent des champs de données (ce sont les variables d'instances) et des champs de méthodes.

La transmission de message utilise les mêmes mécanismes de constructions syntaxiques que le PASCAL ordinaire.

La définition de nouvelles classes se fait en utilisant un nouveau mot clé du compilateur : OBJECT.

Le mécanisme de base est le suivant :

```
TYPE
      Nom Classe = OBJECT (Nom Superclasse)
         <Déclaration de variables d'instance>
         <Définition des entêtes de méthodes>
      END
```

< > dénotent des parties optionnelles.

Les méthodes quand à elles sont définies comme des procédures ou des fonctions du langage PASCAL à la seule condition de les rattacher à une classe définie.

Le schéma de base est :

```
      PROCEDURE NomClasse.NomProcédure (liste d'arguments)
      BEGIN
      .
```

```
.
.
END ;
```

Dans Object Pascal, il n'est pas possible, à l'heure actuelle, de créer des méthodes de classes, des variables de classes, de faire de l'héritage multiple ou d'avoir des métaclasses.

5.2 NEON

Ce langage a été développé par Kriya Systems pour le Macintosh d'APPLE en 1984. La syntaxe de base de Neon est très proche de celle du langage FORTH. Contrairement à Smalltalk où dans une transmission l'objet précède le message et la liste d'arguments, dans NEON, la liste d'arguments et le message précèdent l'objet receveur. C'est un des rares langages objet à se trouver dans cette situation. Nous verrons qu'il en sera de même dans notre extension d'APL.

Les nouvelles classes et méthodes sont définies en utilisant des mots-clés qui délimitent aussi bien les définitions de classe (:CLASS et ;CLASS) et les définitions de méthodes(:M et ;M).

Les schémas de base sont :

```
: CLASS Nom classe<Super Nom Superclass> <n Indexé>
        <noms de variables d'instance>
        <Définitions de méthodes>
; CLASS
et
: M Selector: <{nom arguments\variables locales-résultats}>
        <corps de méthode>
; M
```

< > dénotent toujours des parties optionnelles.

Les arguments nommés dans NEON permettent d'associer des noms aux arguments. Ces derniers sont placés dans la pile avant l'appel d'une méthode ce qui permet de les accéder simplement en faisant référence à leur nom dans le corps de la méthode.

Les variables locales permettent uniquement l'usage de variables temporaires dans le corps d'une méthode.

Notons également que dans NEON on peut choisir entre la liaison statique et la liaison dynamique.

5.3 OBJECTIVE-C

Objective-C a implanté les notions de base des L.O.O au sein du langage C de manière indépendante de toute machine. Ceci est réalisé grace à un compilateur qui accepte un code source Objective-C et fournit un code source C équivalent. C'est en fait un préprocesseur.

Objective-C est un sur-ensemble du langage C. L'extension objet à été réalisée en rajoutant un nouveau type d'expression dans C : l'expression de message. Syntaxiquement, l'expression de message est délimitée par des crochets. La distinction entre les crochets d'indexation de tableaux standards et les crochets d'expression de message, se fait par le contexte.

La syntaxe d'une transmission de message est très voisine de celle Smalltalk.

Les nouvelles classes sont définies dans un fichier spécial de description de la manière suivante :

= Nom Classe : Super Classe (Liste de phyla){Déclarations des variables d'instance}
+ Nom Methode Classe {Implantation de Methode}
- Nom Methode Instance {Implementation Methode}
= :

Il faut noter qu'une seule classe peut être définie dans un tel fichier.

Un des concepts des L.O.O propres à Objective-C est le phyla. Le phyla dans Objective-C est un groupe de classes. Indiquer qu'une classe appartient à un groupe particulier de classes, permet au compilateur de générer efficacement des tables de structure de méthodes.

Une des grandes forces d'Objective-C est sa disponibilité sur une large gamme de machines.

5.4 C++

C++ est une autre extension du langage C, réalisée par Stroustrup en 1984 dans les laboratoires de la compagnie Bell.

Le concept de classe dans C++ généralise le type prédéfini **struct** du C classique aux fonctions appelées alors méthodes. L'introduction du mot clé **public** permet de séparer champs et méthodes en deux catégories, privées et publiques.

Voici un exemple de définition de classe tirée de [Gautron86] :

```
class virtty{
  /* champs et méthodes privées */
           char efface_ligne[LONG_LIGNE]; /* initialiser
                                    par des blancs */

  /* champs et méthodes publics */
  public:
    /*
       l(igne) et c(olonne) par
       rapport au haut gauche de l'écran
       impression et effacement
       avec positionnemenet du curseur
    */
     void ecrire ( int,int,char*);/* ecrire une
                                   chaine en l,c */
     void effacer ( int,int,int);/* effacer n
                                  caracteres en l,c */
     void cleol ( int,int);/* effacer
                             la ligne a partir de l,c */
     void cleos ( int,int);/* effacer le reste
                             de l'ecran a partir de l,c */
}
```

Les définitions de méthodes suivent par ailleurs la syntaxe suivante :

```
   Nom classe.Méthode (liste-arguments){


/* corps de la méthode */


   }
```

Au point de vue syntaxe, l'obligation d'expliciter le type des arguments des méthodes est la principale source d'incompatibilité avec le C classique.

C++ implante. les concepts d'héritage simple et multiple. La transmission de message dans C++ utilise la notation pointée et fléchée de C classique.

5.5 OBJVLISP

Historiquement, ce modèle est né des travaux de Cointe sur Smalltalk-76 [Cointe83a] et d'une synthèse à partir d'une sémantique opérationnelle donnée en LISP [Cointe84a]. C'est une extension objet du langage VLISP.

Les objets OBJVLISP sont implantés comme des atomes Lisp ayant pour valeur fonctonnelle une fermeture auto-référencée.

La syntaxe d'une transmission est :

```
(Send 'UN-OBJET selector Arg1 ... Argn)
```

La définition ci-dessous de UN-OBJET, exposé en LISP, traduit la sémantique d'une transmission.

```
(de UN-OBJET (selector args) -1- (let((oself 'UN-OJBET)(mclass
'UNE-CLASSE)) -2- (letdic(car(f-dico mclass))(f-dico oself) -3- (protect
(apply(lookup mclass) selector) args) -4- (rewrite oself mclass)))))
```

Les champs **oself** dénote le nom de l'objet, le champ mclass celui de sa classe.

La transmission de message ObjVLisp se ramène à l'évaluation d'une forme LISP. Autrement dit, à un simple appel de fonctions. Le couple (eval, apply) est préservé, l'implantation du modèle ObjVLisp ne nécessite donc pas la création d'une évaluation particulière.

Voici un exemple de définition d'une classe PILE :

```
(Send 'CLASS 'new 'PILE '(subclassof : OBJECT) '(fields : head length)
 'init (ld() (setq length(length head)) oself)
 'length (ld() length)
 'empty? (ld() (if head nil))
 'print (ld() (print head) oself)
 'pop (ld()(decr length)(nextl head))
 'push (ld(n)(incr length (newl headn)))
```

La création d'une instance de PILE de nom DOROTHEE s'obtient par ;

```
(send 'PILE 'new 'DOROTHEE nil 0) = (send (PILE 'new 'DOROTHEE)
'init)
```

Contrairement à Smalltalk-80, ObjVLisp définit une seule métaclasse de nom CLASS sur le modèle de laquelle sont engendrées toutes les autres classes.

Ce modèle permet également l'héritage multiple.

5.6 OBJECT LOGO

C'est un langage orienté objet très particulier. Il est développé par Coral Software Corporation pour le micro-ordinateur Macintosh d'APPLE.

La différence fondamentale entre ce langage et les autres est qu'il n'existe pas de distinction entre classes et instances.

Ainsi dans OBJECT LOGO, tous les objets sont traités de manière uniforme. On peut définir "au vol" des variables et des méthodes d'instance au cours d'une session interactive.

A cause de la simplicité de ces concepts, OBJECT LOGO constitue une extension du langage LOGO par ajout de quelques nouvelles primitives comme :
KINDOF anObject : qui crée un nouvel objet qui hérite de anObject ;

TALKTO anObject : fait de anObject l'objet courant.

En effet, dans OBJECT LOGO, à chaque instant, il existe un objet courant, durant l'exécution toutes les références aux variables et aux procédures se résolvent dans le contexte de cet objet courant.

Il existe des primitives comme HAVE, HOWTO, USUAL, pour ajouter des variables ou des méthodes d'instances, invoquer des méthodes héritées.

L'héritage multiple est disponible sous ce langage.

5.7 OBJECT ASSEMBLER

C'est un jeu de macro pour le langage assembleur Motorola 68000. Il a été développé pour Apple et sera disponible fin 1986.

Les exemples qui suivent montre l'usage de certaines macros.

```
MACRO
ObjectDef &TypeName,&Heritage,&FieldList,&MethodList
```

permet de définir une nouvelle classe. Par exemple :

```
ObjectDef Shape,Object,
    ((boundRec,8), (borderThickness,2), (color,2)),
    ((Draw), (MoveBy), (Stretch))

      ObjectDef Arc,Shape,
            ((startAngle,2), (arcAngle,2)),
```

```
((Draw,OVERRIDE), (GetArea), (SetArAngle))
```

- Définition d'une méthode

```
MACRO
&ProcName ProcMethOf &TypeName

MACRO
EndMethod
```

Exemple

```
Draw ProcMethOf Arc
<code>
EndMethod
```

- Accès à une variable d'instance

```
MACRO
ObjectWith &TypeName

MACRO
EndObjectWith
```

Exemple

```
ObjectWith Arc
MOVE.L startAngle(A1),-(SP)
PEA boundRect(A1)
EndObjectWith
```

OBJECT ASSEMBLEUR supporte l'héritage multiple. Une possibilité intéressante est la définition des sous-classes de OBJECT ASSEMBLEUR, dans le langage Objet Pascal que l'on a déjà évoqué.

5.8 EXPERCOMMONLISP

Ce langage, développé par ExperTelligence, dérive d'ExperLisp disponible sur Macintosh depuis 1985.

La syntaxe d'ExperCommonLisp est très influencée par celle du langage LISP : la transmission de messages, l'initialisation et l'accès aux variables d'instance, sont réalisés grâce à des fonctions LISP.

```
(setq Triangle (Send Object 'Subclass))
```

définit une sous-classe de Object, de nom Triangle, par envoi du message subclass à la classe Object.

```
(setq tr1 (Send Triangle 'New))
```

crée une nouvelle instance de la classe Triangle de nom tr1.

Le schéma complet de la définition d'une classe fournit la possibilité d'avoir des variables et des méthodes d'instance et de classe.

```
  (setq New Class (CLASS (superclass 1 superclass 2 ... superclass n)
(IVS (iv1)(iv2 ... (ivn))
   (method1 (arg-list)(body))
   (method2 (arg-list)(body))
   .
   .
   .
   (methodn (arg-list)(body)))
(CVS (iv1)(iv2) ... (ivn)
(Metamethods (method1 (arg-list)(body))
          (method2 (arg-list)(body))
          .
          .
          .
          (methodn (arg-list)(body))))
```

où IVS, Methods, CVS sont des mots clés pour définir respectivement les variables d'instances, les méthodes et les variables de classe. Enfin, Metamethods est un mot réservé pour la définition de méthode reconnue uniquement par la classe et non ses instances.

On peut remarquer que dans ExperCommonLisp on peut définir plusieurs superclasses d'une même classe (héritage multiple).

5.9 CEYX

Plus qu'un langage, CEYX est un environnement de programmation developpé par l'INRIA. Conçu au départ comme un outil auxilliaire de conception de circuits VLSI, il permet la manipulation de hiérarchies tels que : bases de données (systèmes de fichiers, bibliothèques, etc.), documents (chapitres, sections, paragraphes, lignes, caractères, etc.).

CEYX est une extension de LISP qui offre à l'utilisateur un ensemble de fonctions LISP, permettant de créer et de manipuler des objets. Ces objets résultent de la combinaison d'une structure de type enregistrement avec un ensemble de propriétés sémantiques. Les objets sont regroupés en famille de manière hiérarchique à la SIMULA de sorte qu'ils héritent des propriétés de leurs ancêtres.

Un enregistrement est un objet composé de champs particuliers appelés valeurs. Par exemple, la définition d'un objet fenêtre de nom RECT se fait de la façon suivante :

```
(defrecord RECT xorg yorg
        xdim ydim
        couleur)
```

Cette information est utilisée par le mécanisme d'instanciation de la manière suivante :

```
(setq moncarre
        (rmakeq RECT xorg 0 yorg 0)
(rputq RECT couleur moncarre 'vert)
```

La construction DEFSEM sert à relier une propriété d'affichage de texte à la structure RECT :

```
(defsem (RECT affichage) (texte rectangle)

                    < corps de la fonction >
```

L'activation d'une instance s'obtient par l'utilisation de la construction SEND :

```
(send 'affichage <instance> <texte>
```

Chaque fois qu'une nouvelle famille d' objets est définie, pour l'éditer CEYX utilise un mécanisme élémentaire, qui peut être redéfini par ajout de propriétés particulières à ces objets. Ce qui permet à l'utilisateur d'étendre au maximun les possibilités de l'éditeur.

CHAPITRE 4

Chapitre 4

OBJAPL 90 : UN APL ORIENTE OBJET

1 MOTIVATIONS

APL est très performant dans le traitement des données et des expressions :

- les structures de données sont claires, consistantes et puissantes ;
- le mécanisme des expressions est algébrique et fonctionnel.

En revanche dans le domaine des types abstraits, APL et la théorie des tableaux de T. More [More73] sont très pauvres.

Le fait qu'il n'y ait que deux types de données (numériques et caractères) entraine une simplicité certaine, mais qui semble aujourd'hui quelque peu anachronique. En effet, la plupart des langages de programmation proposent de fournir des mécanismes qui consistent non plus à déclarer ses variables mais à permettre à l'utilisateur de définir ses propres types de données grace au concept d'abstraction des données.

Lorsqu'on examine les diverses propositions dans ce domaine, on s'aperçoit très vite que deux extensions sont désirables dans APL :

- définition des types de données par l'utilisateur

- Structuration de l'espace global des noms en des unités plus petites

Plusieurs tentatives ont été faites dans ces voies. Les langages X\APL [Braffort] et ALICE [Jenkins80] permettraient la définition de nouveaux types à travers des mécanismes de marquage des structures. On peut également définir plusieurs versions de fonctions qui s'appliquent à ces structures.

Hardwick [Hardwick81] utilisa la notion d'enregistrement pour typer les structures de données dans une application graphique.

Kajiya [Kajiya81] propose un nouveau mécanisme de portée des fonctions afin de les rendre génériques.

Orgass [Orgass77] propose l'introduction dans le langage du concept d'espace nommé. Ce n'est pas une zone de travail, mais une entité qui contient des fonctions, des variables, une pile d'état, la communication entre ces entités se faisant à travers des variables partagées. Ce mécanisme permettant selon lui l'introduction des données de structure quelconque dans APL.

Jusqu'à présent aucune de ces propositions n'a été largement approuvée par la communauté APL. Les "APListes" pensent qu'un typage fort des données est étranger à l'esprit d'APL (ce fut la même attitude face aux structures de contrôle FOR, IF, THEN, ELSE, DO, WHILE etc ...).

C'est pourquoi seule une solution élégante qui n'altère ni les concepts, ni l'esprit du langage, qui soit simple et pratique, peut mettre fin à ces réticences. C'est ce que nous nous proposons de définir.

Dans APL, le seul mécanisme d'abstraction reste de nos jours la portée dynamique des fonctions. Ce concept de fonction souffre par ailleurs d'un certain nombre de problèmes :

- Il ne permet pas de gérer de manière adéquate la complexité. En effet, les grandes zones de travail APL sont très souvent des collections non structurées d'un nombre important de fonctions.

- Les fonctions ne séparent pas la représentation du comportement. Bon nombre de programmes APL nécessitent des modifications dès qu'un changement intervient dans la représentation externe des données.

Nous allons introduire de nouveaux mécanismes d'abstraction dans le langage et montrer que cela n'affaiblit ni l'aspect fonctionnel, ni le style de programmation d'APL. Ces mécanismes sont ceux inspirés des L.O.O. notamment de Smalltalk. Ce qui nous permettra dans APL :

- de définir non seulement des types de données qui paraissent naturelles pour une application donnée, mais aussi les fonctions adéquates qui doivent les traiter.

- d'introduire une définition hiérarchique des types de données (notion de sous-classe)

- d'associer à des fonctions différentes des noms identiques dès lors qu'elles sont s'adressent à des classes distinctes. Le filtrage assure que la fonction adéquate (ou méthode) sera toujours sélectionnée.

Dans APL les tableaux sont des objets "de première catégorie" ("first Class Citizen", selon l'expression consacrée) dans la mesure où on peut les manipuler de manière non ambiguë (accès à leurs caractéristiques etc ...).

APL unifie les entiers et les flottants en ce sens que l'on utilise par exemple le même symbole + pour représenter l'addition pour ces deux types primitifs. Ce n'est pas le cas des fichiers, pour la désignation desquels on utilise d'une part des nombres à défaut de pouvoir les nommer, d'autre part les fonctions d'accès (FREAD, FWRITE...) ne portent pas le même nom selon que l'on s'adresse aux fichiers séquentiels ou non. L'extension objet peut servir à unifier les catégories d'objets dans APL.

A. Kay disait lors du congrès APL 81 [Kay81] que trois voies différentes dans le développement des concepts génériques ont été suivies par APL, Algol 68, et les langages objets comme SIMULA et Smalltalk. Il indiquait en particulier qu'APL avait été l'une des premières et des plus importantes applications des mécanismes de généricité, mais que ce langage ne faisait plus aujourd'hui qu'un usage très faible de ces idées.

APL a-t-il failli être un langage objet ?

En effet, une certaine similitude existe. Dans les L.O.O, il n'existe qu'une seule structure de donnée : **l'objet.** En APL, il n'existe qu'une seule structure de donnée : **le tableau.** Dans les L.O.O, il n'existe qu'une seule structure de contrôle : **la transmission de messages.** En APL, il n'existe qu'une seule structure de contrôle : **l'application fonctionnelle.**

Alors que manque-t-il à APL pour être un L.O.O ?

Nous répondrons : tout simplement un certain nombre de concepts qui s'appellent : **objet, classe, instanciation, héritage, transmission de messages.** Ces concepts sont donc caractéristiques d'un langage objet. Nous allons dans la suite de cet exposé proposer quelques idées sur ces différents points en vue d'une extension d'APL orientée objet.

Nous nous inspirerons du formalisme Smalltalk qui s'énonce en cinq postulats [Cointe85] :

P 1 : chaque entité du langage est uniformément considérée comme un objet ;

P 2 : l'unique structure de contrôle est la transmission de messages ;

P 3 : chaque objet appartient à une classe qui détermine son comportement fonctionnel et les attributs statiques le caractérisant. Plus précisément, une classe est un objet générateur capable d'engendrer d'autres objets appelés ses instances. Suivant la philosophie "platonicienne" tous les objets créés sur son modèle ont la même forme mais sont reconnaissables par les valeurs de leurs attributs communs ;

P 4 : une classe est un objet à part entière engendré sur le modèle de la méta-classe CLASS ;

P 5 : il existe une hiérarchie des classes, toute classe définie comme sous classe d'une autre hérite de ses fonctionnalités (les méthodes) et de ses attributs (les champs).

L'analyse détaillée de ces postulats, la recherche systématique de leur adéquation à la philosophie et au style du langage APL, nous conduira à la définition d'un modèle objet pour notre système APL 90, que nous appelerons OBJAPL 90.

De manière générale, nous pensons qu'APL peut bénéficier de la plupart des avantages de la programmation orientée objet. A savoir :

- Encapsulation
- Modularité
- Accès unifié aux objets tels que les fichiers (comme préconisé par [Crick81])
- Accès à de nouveaux types de données
- Réutilisation et partage de code

Ce dernier aspect paraît très pratique. Une application APL qui utilise des éléments d'une bibliothèque, doit les copier dans la zone de travail active. Ce qui présente bon nombre d'inconvénients :

- Le coût de l'opération de copie peut être très élevé notammment si celle-ci est fréquente.
- Si l'application n'est pas capable d'assumer elle même la copie, alors il y a lieu de dupliquer le code des éléments non seulement dans l'application, mais également dans toute zone de travail où cette application est utilisée.
- Des problèmes de cohérence de versions se posent.

Nous voulons une extension orientée objet qui réponde au mieux aux critères que nous avons définis dans la première partie. Pour une bonne intégration de l'extension orienté objet dans l'univers APL, il faut qu'elle soit guidée par quelques principes :

- ne pas entraîner d'altération de la syntaxe et de la méthode d'interprétation du langage APL ;

- conserver les qualités procédurales ou fonctionnelles d'APL ;
- ne pas reposer sur le postulat P1 que nous ne pourront pas respecter. Car contrairement à Smalltalk qui est un langage objet par essence, OBJAPL 90 est un langage hybride conventionnel, étendu par des concepts de la P.O.O. Le système étendu doît être mixte, en ce sens que tout n'est pas objet (c'est d'ailleurs le cas pour beaucoup de L.O.O).
- ne pas soumettre l'extension aux fonctionnalités d'un modèle objet particulier. En effet, à chaque langage peut correspondre une vision objet appropriée. Bien que nous proposions une extension avec la vision Smalltalk, nous nous inspirerons de ce modèle uniquement parce qu'il permet d'exhiber de manière claire et simple les mécanismes qui régissent les L.O.O. L'objectif dans un premier temps est d'avoir un système expérimental.

2 UN MODELE OBJET POUR APL : OBJAPL 90

2.1 LE CONCEPT OBJET DANS OBJAPL 90

La question à laquelle il faut répondre est : quel statut doivent avoir les objets dans APL?

Pour respecter la philosophie d'APL, nos préoccupations seront proches à la fois de [More79], [Hassit79], [Jenkins81], [Brown81] et de [Kajiya83]. Pour l'évaluation de nos propositions, nous essayerons de mettre en oeuvre les critères que nous avons élaborés dans le chapitre 2, notamment les règles d'équivalence d'APL qui permettent de comprendre aisément le comportement d'une primitive.

La première tentation est d'essayer de donner aux objets le statut de tableau, car c'est la seule structure de donnée d'APL. L'avantage espérée de cette approche est que les objets bénéfieront ainsi de toute la puissance des fonctions et des opérateurs primitifs d'APL.

Soit F une fonction APL et T un tableau APL, E une expression, IL une liste d'index, AX est un numéro d'axe. Observons le résultat de certaines constructions :

F T (1)
T F T (2)
T T (3)
TD <- T (4)
T[IL] (5)
T[IL] <- T (6)

Les constructions (5,6) par exemple ne s'appliquent pas à des scalaires. Elles représentent respectivement l'indexation, l'affectation indicée. Ainsi faut-il encore préciser si un objet est un tableau scalaire ou non ?

Une analyse attentive fait apparaître rapidement que le statut de tableau APL ne convient pas aux objets. En effet si ***A*** est un objet tel que :

```
      ρA
3
```

il serait légitime que les expressions suivantes : `2↑A`, `A[1]` et `⌽A` soient soient consistantes eu égard à ce que l'on obtiendrait si ***A*** était un vecteur à 3 éléments. Autrement dit, les identités suivantes doivent être vérifiées :

```
      2↑A  ↔ A[1],A[2]
      A[1]↔ 1↑A
      ⌽A ↔ A[3],A[2],A[1]
```

Malheuresement la plupart des objets que nous souhaitons définir seront complexes et ne pourrons pas respecter ces identités.

Par exemple nous avons tenté d'introduire dans APL 90 une nouvelle structure de donnée qui était la table de symboles, nous avons constaté certaines inconsistances. Une table de symboles est une structure de donnée qui contient un ensemble non ordonnée d'éléments désignés par des noms (symboles). On peut la considérer comme un "package" au sens d'APL-SHARP. Dans APL 90, une table de symboles peut contenir un nombre quelconque d'éléments et est utilisée pour représenter l'organisation des données dans l'espace virtuel.

Une fonction monadique comme ρ aurait pu être un bon candidat pour fournir le nombre d'éléments d'une telle table. Si notre table T contient 3 éléments de noms ***ANNE***, ***OUAGADOUGOU*** et ***MICHEL***, `ρT` pourrait répondre 3. Cependant, cela n'implique pas que `T[1]`, `T[2]`, et `T[3]` sont des expressions valides pour désigner ces éléments, et à l'heure actuelle, il serait très difficile d'implanter une indexation efficace de ces tables. Il paraît également difficile de trouver un sens approprié à des expressions comme `2↑T` ou `⌽T`. Par aileurs l'usage des fonctions primitives APL font apparaître des inconsistances similaires.

Ces considérations illustrent clairement que les objets que nous allons définir, ne doivent pas se comporter comme des tableaux au sens APL, et que le statut le plus approprié pour un **objet** est d'être une **entité atomique.**

Bon nombre de langages de programmation ont tendence à confondre les concepts de **type de donnée** et de **structure de donnée.** Un exemple significatif est le langage PASCAL dans lequel 1 type est renommé type, une structure, type aussi, etc... Il nous semble important de noter ce qui les différencie.

Un type de donnée n'a pas de structure. Sa représentation peut être inconnue de l'utilisateur. En APL par exemple l'utilisateur ne peut pas accéder directement au format en virgule flottante des nombres réels. Seul le comportement des réels lors d'opérations arithmétiques et leur visualisation intéresse le programmeur. Un type de donnée est atomique.

Une structure de données n'est pas atomique. En APL par exemple, le rang, les dimensions, et le type de chaque élément d'un tableau sont accessibles en lecture par l'utilisateur. APL attache une grande importance à la notion de structure de donnée, alors que la notion de type de donnée y était relativement floue. Depuis que dans APL, les tableaux hétérogènes ont fait leur apparition, la distinction est apparue clairement.

APL 90 insiste sur la distinction entre structure de donnée et type de donnée. Le terme de structure caractérise un arrangement d'éléments, ainsi que les mécanismes à utiliser pour accéder à un élément ou à un groupe d'éléments de cet arrangement. Le terme type en revanche caractérise l'appartenance d'une entité à un ensemble d'éléments réunies par des propriétés communes. Ainsi,

```
A ← 'ABC'

B ← 3  4  5

C ← 8.785
```

les variables ***A*** et ***B*** ont même structure, en ce sens que leurs éléments sont arrangés sous la forme d'un tableau de rang 1, et de dimension 3.

De même, les entitées contenus dans les variables B et C appartiennent au même type, qui est celui des entitées numériques.

Dans OBJAPL 90, il est fondamental d'insister sur la différence entre un objet (au sens type de donnée) et une structure de donnée :

1. Un objet défini n'a pas de structure, mais est atomique.
2. Un scalaire est une structure de données contenant un objet (primitif ou défini) unique. Le rang d'un scalaire est 0, sa dimension est un vecteur vide. Sa profondeur dépend de l'objet qu'il contient.
3. Un tableau est une structure de données contenant 0, 1, ou plusieurs objets. Le rang d'un tableau est un nombre positif ou nul, indiquant le nombre de ses dimensions. Un scalaire est un cas particulier de tableau.

En particulier une variable APL, désigne toujours une structure de donnée, qui peut être un scalaire et contenir un objet, mais jamais l'objet lui même. La formulation " l'objet X" doit être comprise comme : " la variable X désignant un certain scalaire contenant l'objet qui nous intéresse ... ".

Plus généralement, toute opération "rendant un objet" retourne en fait une structure de donnée contenant l'objet en question. Ainsi le nombre 3 est un objet atomique , mais l'expression APL :

```
3
```

calcule un scalaire dont le contenu est le nombre 3.

Notons enfin que l'on ne manipule pas des objets, mais les réprésentations de ces objets. Désormais quand nous parlerons d'objets, il s'agit d'objets définis (au même sens qu'il existe des fonctions définies), car si un objet est un type de donnée, on peut qualifier d'objets primitifs les types numérique et caractère d'APL.

Signalons au passage que dans ObjVLisp, les objets sont implémentés comme des atomes LISP.

Un **objet** est donc un nouveau **type de donnée** qui peut être défini par le concepteur ou l'utilisateur du système APL. Un objet doit avoir des propriétés similaires à celles des types de données primitifs d'APL. Un objet étant encapsulé dans un scalaire, il doit se comporter comme tout autre scalaire simple, par conséquent si *OBJ* est un objet, les identités suivantes sont vérifées :

```
OBJ ≡ ⊂OBJ           ⍝ ENCLOSE ET DISCLOSE N''ONT
OBJ ≡ ⊃OBJ           ⍝ PAS D''EFFET SUR UN OBJECT
0 = ≡OBJ             ⍝ SA PROFONDEUR EST 0
(⍳0) ≡ ⍴OBJ          ⍝ C''EST UN SCALAIRE
```

Un point de vue pragmatique est de dire que la définition des objets est à OBJ APL90 ce que la définition des opérateurs est à APL 2.

2.2 LA TRANSMISSION DE MESSAGE DANS OBJAPL 90

Le postulat P2 conduit à définir un protocole unique de communication qualifié de transmission de messages.

Dès lors se posent les problèmes de choix syntaxiques. Certains L.O.O qui sont des extensions de langages procéduraux ou fonctionnels existants ont choisi des syntaxes proches du langage de base, d'autres ont opté pour une syntaxe nouvelle ou proche de Smalltalk.

A titre d'exemple, voici la syntaxe de quelques L.O.O. Le message **msg** avec l'argument **arg** est envoyé à l'objet référencé par **obj.**

Syntaxe	Langage
obj msg: arg	Smalltalk
obj.msg (arg);	Object Pascal
arg msg: obj	Néon
(obj'msg<arg>)	Exper Common LISP
[obj ms:arg] ;	Objective-C
MOVE.W arg(A6),-(SP)	Object Assembleur
MOVE.L obj(A6),-(SP)	
Meth call msg	
tell ; obj ; [msg "arg]	Object Logo
(Send'obj msg arg)	ObjVLisp

De notre point de vue la syntaxe dans une extension orientée objet doit rester la même que dans APL standard. En d'autres termes, il ne doit pas y avoir de préfixage des lignes qui font référence à l'extension objet comme c'est le cas dans Objective C. La syntaxe dans un APL orienté objet doit être compatible APL-ISO, ce qui permet de prendre appui sur les structures de base d'APL. Ceci évite par ailleurs de faire un collage de la syntaxe d'un langage objet existant (Smalltalk par exemple) et une transcription dans la bonne syntaxe du langage cible. Un autre intérêt est qu'il n'est pas nécessaire d'avoir deux machines d'évaluation, une pour le langage de départ et une autre pour son extension objet.

Constatons tout d'abord que

```
3+4
```

est une expression valide aussi bien pour APL que pour Smalltalk et calcule la même valeur qui est 7. On peut donc essayer d'établir un parallèle entre la syntaxe de Smalltalk et celle des appels de fonction en APL.

Dans Smalltalk, tout objet est actif à la réception d'un message. Nous désignerons par RECEVEUR l'objet à qui on a envoyé un MESSAGE. Une transmission dans Smalltalk peut être représentée par :

TRANSMISSION := <RECEVEUR MESSAGE>

MESSAGE := < SELECTEUR {PARAMETRES} >

La syntaxe d'envoi d'un message à un objet (OBJ) est :

OBJ MESSAGE

Que doit-elle être dans APL ?

Examinons tout d'abord le comportement des fonctions primitives monadiques :

```
                M ← 9 4 6 8 2
⍴M

   5
```

signifie que l'on applique la fonction ⍴ à l'objet M. La vision objet peut consister à dire que l'on envoie à l'objet M le message composé du sélecteur ⍴, et que celui-ci répond en retournant la valeur d'un de ses champs.

Cette vision peut en fait s'appliquer uniformément à toutes les fonctions primitives monadiques. Leur syntaxe suggère donc la forme suivante pour une transmission en APL :

TRANSMISSION := <MESSAGE RECEVEUR>

Intéressons nous maintenant aux primitives diadiques. Les fonctions de restructuration suggèrent elles aussi la forme :

TRANSMISSION := <MESSAGE RECEVEUR>

Ainsi :

```
5⍴M
```

signifie que *M* reçoit le message `5⍴` composé du sélecteur `⍴` et du paramètre 5.

```
3 ↑ V
```

est l'envoi du message `3 ↑` à l'objet *V*

Il en est de même pour les fonctions primitives scalaires diadiques.

Ainsi : `3×5` est une demande à `5` de se multiplier par le paramètre `3`.

```
2 1 3⍉X
```

peut être vue comme un envoi à l'objet *X* du message de sélecteur `⍉` et d'argument `2 1 3`.

Donc dans une extension objet d'APL on aura :

TRANSMISSION := <MESSAGE RECEVEUR>

MESSAGE := <{PARAMETRES} SELECTEUR>

L'exécution d'APL se faisant de droite à gauche, la syntaxe d'envoi d'un message à un objet APL s'exprimera sous la forme :

par1 ... parn   sélecteur   objet-receveur

On voit y apparaitre : le nom de l'objet à activer (le receveur), précédé du nom de la méthode à activer que précède la suite des paramètres éventuels.

La ligne APL :

```
A - B + C ⋆ D
```

s'exécute donc ainsi : l'objet `D` reçoit le message `C ⋆`, le résultat T1 reçoit le message `B +`, le résultat T2 reçoit le message `A -`.

Les objets peuvent recevoir des messages définis par l'utilisateur compatibles avec la syntaxe précédente, par exemple il semble naturel d'ecrire des expressions telles que :

```
OPEN MYWINDOW
OPEN MYFILE
100 350 MOVE MYWINDOW
(2 READ MYFILE) DISPLAY MYWINDOW
CLOSE MYFILE
CLOSE MYWINDOW
```

Ce fragment de programme manipule deux objets l'un étant une fenêtre sur un écran graphique et l'autre un fichier situé en dehors de la zone de travail active. On peut faire remarquer la généricité des méthodes ***OPEN*** et ***CLOSE***.

Les deux nouvelles formes syntaxiques que l'extension objet impose de reconnaitre sont les suivantes :

```
TRANSMISSION := METHODE     OBJET     |
                PARAMETRES     METHODE     OBJET
```

Elles peuvent s'interpréter comme l'appel de la fonction "METHODE" associée à l'objet receveur.

2.2.1 Les sélecteurs de message définis par l'utilisateur

Dans les exemples ci-dessus, les sélecteurs de message sont des fonctions primitives APL ou des fonctions définies mais, il est fort probable que les primitives ne soient pas adaptées aux objets que nous serons emmenés à définir. En effet si WINDOW1 et WINDOW2 sont deux objets de type fenêtre, que signifier signifie l'expression

WINDOW1 ⋆ WINDOW2

Faut-il surcharger la signification de ⋆ en lui donnant un sens particulier dans le contexte de l'objet de type fenêtre ?

Une courte expérience fait apparaître certaines inconsistances. Il est bien connu qu'une ligne APL telle que :

A B C

peut avoir différentes interprétations syntaxiques en fonction des valeurs de A, B et C. Un nouveau problème survient si ***C*** est un objet. La valeur de ***B*** doit-elle être recherchée dans la zone de travail active, ou doit-on considérer B dans tous les cas comme un sélecteur de message pour C ? Si ***B*** est un sélecteur doit-on considérer ***A*** comme paramètre du message ? Si on admet des sélecteurs ambivalents comment distinguer un usage monadique du sélecteur ***B*** suivi d'un usage monadique du sélecteur ***A***, d'un usage diadique de ***B*** avec ***A*** comme paramètre ? Que faire si ***A*** et ***B*** sont définis dans la zone active ? Comment faire de la "strand notation" avec les objets ?

Par ailleurs, dans les réalisations actuelles de système APL, si ***C*** est reconnu comme un objet, ***A*** et ***B*** seront probablement toujours liés à leurs valeurs dans la zone de travail active.

Du reste que faire si l'utilisateur désire appliquer une fonction ***F*** à un objet ***O***? Que doit-être la syntaxe ? Doit-on interpréter ***F O*** comme l'envoi du message ***F*** à l'objet ***O***, plutôt que l'application de la fonction ***F***?

L'introduction des objets nécessite d'apporter des réponses à ces questions.

On peut envisager le scénario suivant : si à l'analyse on rencontre un objet, on change de contexte ; le sélecteur de la méthode qui suit l'objet est recherché dans le contexte de cet objet. En cas d'echec, on peut décider de continuer ou non la recherche dans le contexte de la WS active.

Cette situation crée quelques problèmes dans un univers APL. L'intérêt de la P.O.O est de pouvoir enrichir son environnement de départ en créant ses propres objets (objets définis) et ses propres méthodes (méthodes définies). Citons quelques exemples qui mettent en évidence les problèmes qui se posent.

Soit l'expression

OPEN WINDOW2

deux cas de figure existent :

- on recherche la méthode OPEN dans la WS active, si elle n'y est pas définie alors il y a apparition d'un rapport d'erreur :

```
VALUE ERROR
OPEN WINDOW2
^
```

Si OPEN y est défini, cette fonction s'appliquerait aussi bien sur WINDOW2 que sur toute autre variable de la WS active. Il n'y a par conséquent aucune encapsulation, aucun lien à priori entre WINDOW2 et OPEN.

- on recherche OPEN parmi les comportements attachés à la classe de l'objet WINDOW2. Il y a changement de contexte. Dès lors apparaît un autre problème qui est "l'imcompatibilité avec l'existant en APL". En effet, dans l'expression suivante :

```
OPEN¨ WINDOW1  WINDOW2
```

WINDOW1 sera considéré comme un sélecteur de méthodes et recherché dans le contexte de WINDOW2. Or manifestement l'opération espérée était de faire d'abord un vecteur généralisé avec WINDOW1 et WINDOW2 puis d'appliquer à chaque élément de ce vecteur enclos la méthode OPEN.

Un autre exemple :

```
PRINT A  B DISPLAY   WINDOW1
```

On passe dans le contexte de WINDOW1, on y trouve DISPLAY puis on revient à la WS active pour rechercher le paramètre du message DISPLAY. Mais celui-ci est-il B, A B ou PRINT A B ? Alors que manifestement la sous expression

```
A  B DISPLAY WINDOW1
```

rend comme résultat un objet temporaire de même type que WINDOW1.

Les objets définis ne sont donc pas des variables APL ordinaires.

Pour résoudre ces problèmes il faut pouvoir indiquer de manière spécifique, les identificateurs qui sont des sélecteurs de message. Dès lors on s'attaque à l'un des principes auxquels les "APListes" tiennent : "pas de mots reservés dans le langage". Pour contourner ce principe, nous proposons l'introduction d'identificateurs **distingués** pour les **sélecteurs** de message (notons que cela existe déjà : les fonctions et les variables systèmes qui commencent par un ⎕). Nous avons choisi arbitrairement le signe **:** (deux points) comme premier symbole d'un identificateur de sélecteur de message. Cette proposition n'est en rien définitive et pourrait être modifiée dans les versions ultérieures de notre réalisation. ***:PRINT***, ***:CLOSE***, ***:OPEN*** et ***:SHOW*** sont des exemples de tels sélecteurs de méthodes.

Du point de vue de l'utilisateur, un sélecteur se comporte comme une fonction ambivalente :

```
      F←+ ◇G←:CLOSE
      ⎕NC ⊃'F ' 'G'
3 3
```

Notons qu'un sélecteur n'est jamais appliqué directement sur un objet, mais sur le scalaire dans lequel l'objet est encapsulé. L'expression

```
      2 3 5 :ADD OBJ
```

envoie à `OBJ` un message composé du sélecteur `:ADD` et du paramètre `2 3 5`.

Ainsi à chaque fois qu'à l'analyse on rencontre un identificateur précédé de `:`, on active le contexte de l'objet auquel il se rapporte (l'objet courant). Sinon, on reste dans le contexte de la WS active. En d'autres termes les sélecteurs de messages seront des **points d'entrées** dans le contexte d'une classe.

Un autre choix syntaxique important est l'ambivalence des méthodes (comme pour les fonctions primitives APL). En effet la plupart des symboles APL admettent une forme monadique et une forme diadique qui correspondent à des fonctions différentes. Par exemple `!` en monadique signifie la fonction factorielle, en diadique c'est le coefficient binomial. Avec l'apparition des nouveaux APL de type APL2 d'IBM, on admet que les fonctions définies soient également ambivalentes. Une méthode peut être une fonction définie. La cohérence et la compatibilité impliquent l'ambivalence des méthodes.

2.2.2 Les fonctions primitives face aux objets

On utilise les même signes `- + × =` pour représenter respectivement les fonctions usuelles de soustraction, d'addition, de multiplication et d'égalité aussi bien pour les entiers que pour les flottants. A chaque type correspond un algorithme particulier. La prise en compte des différents types de données est transparente à l'utilisateur et lui échape complètement. Dans la plupart des systèmes ces symboles sont rattachés à des fonctionnalités bien précises et définies une fois pour toute. L'utilisateur n'a aucun moyen de modifier le sens qui sont attachés à ces symboles.

Pour fournir un environnement de plus en plus convivial, il est très tentant de proposer la redéfinition des symboles primitifs d'APL. Autrement dit, la possibilité pour l'utilisateur de donner le sens qu'il **désire** aux symboles primitifs du langage natif.

Malheureusement les fonctions primitives ont un comportement très fortement liés à la structure des données. Les fonctions primitives scalaires par exemple sont pénétrantes. Pour s'appliquer elles traversent toutes les couches de la structure de donnée pour atteindre les éléments terminaux. Ce mécanisme est implicite dans

APL. Or les méthodes ne sont pas pénétrantes à priori (comme les fonctions définies). Si on autorise la redéfinition des symboles primitifs d'APL, des inconsistances apparaissent. En effet les symboles primitifs seront pénétrants quand elles seront liées à des fonctions primitives et non pénétrantes quand elles sont définies comme sélecteurs de méthodes.

Par ailleurs cette redéfinition n'est pas possible pour tous les symboles primitifs d'APL, vu le rôle fondamental que jouent certains dans la théorie des tableaux qui est le fondement du langage APL. En effet on sait que les résultats de ρ ou de ≡̲ denotent des propriétés fondamentales pour toute donnée APL, les redéfinir sur les objets enleverait toute consistance à l'extension orientée objet. On ne peut donc pas redéfinir les symboles primitifs d'APL.

Cependant les fonctions primitives opèrant sur des tableaux APL, les objets étant encapsulés dans des scalaires, et un scalaire étant un tableau APL, il est légitime que les fonctions primitives opèrent sur de tels scalaires que nous appelerons objets par abus de langage. La question à resoudre est la suivante : les fonctions primitives doivent-elles opérer activement sur les objets de manière nominative en leur envoyant des messages, ou passivement c'est à dire en manipulant simplement leur référence sans aucune interaction avec l'objet ?

La réponse pourrait être faite au cas par cas, mais APL lui même peut nous aider confortablement. En APL on classe habituellement les fonctions primitives en trois catégories :

- les fonctions scalaires
- les fonctions de restructuration
- les fonctions mixtes

Les fonctions primitives scalaires, reconnues également comme des fonctions pénétrantes, opèrent réellement à travers les structures de leurs arguments jusqu'aux éléments les plus profondément encapsulés dans la structure. Ces fonctions peuvent par conséquent opérer sur des objets (ou plus généralement sur des tableaux contenant des objets) en leur envoyant des messages et en collectant les résultats dans de nouveaux tableaux.

Dès lors quel message doit-on envoyer aux objets ? Nous utiliserons l'opérateur monadique *each deep* ↓̈ (proposé par différents auteurs), comme mécanisme explicite qui expliquerait le comportement des primitives scalaires. Cet opérateur est également connu comme *opérateur d'extension scalaire,* qui transforme une fonction quelconque en sa forme pénétrante.

Nous considérerons qu'une fonction primitive scalaire telle que + est en fait une fonction dérivée définie comme :***ADD***↧, qui est la composition du message :***ADD*** avec l'opérateur d'extension scalaire. Donc l'application d'une fonction primitive monadique ou diadique sur une structure, consiste à envoyer le message correspondant à chaque objet contenu dans la structure. Par exemple,

```
2 3 4 + OBJ
```

envoie à ***OBJ*** trois messages, de sélecteur :***ADD***, et de paramètres respectifs 2, 3 et 5.

réciproquement, tout message peut être transformer en une fonction pénétrante :

```
CLOSE ← :CLOSE ↧
```

et être utilisé comme suit :

```
CLOSE FILE1 FILE2 FILE3
```

Le cas des fonctions primitives de restructuration est simple : elles créent des nouveaux tableaux à partir de leurs arguments par une simple duplication des références des objets. En cas de remplissage complétif il y a utilisation de prototype (nous étudierons plus loin la notion de prototype dans le cas d'un objet).

En ce qui concerne les fonctions mixtes, elles sont définies comme des algorithmes spécifiques qui opèrent aussi bien sur la structure que sur les valeurs de leurs opérandes. Un tel algorithme peut être écrite sous forme de fonction définie par l'utilisateur, et composée uniquement de fonctions scalaires ou de restructuration : ce qui suffit pour décrire leur comportement sur les tableaux contenants des objets. Par exemple, la fonction ⊥ peut être décrite par un algorithme voisin de celui du produit interne utilisant les fonctions primitives + et ×; en cas d'application aux objets ⊥ doit envoyer aux objets des messages tels que : :***ADD*** ou :***MULT***.

Pour finir notons que d'autres fonctions peuvent envoyer des messages spécifiques. C'est par exemple le cas de ⍕ qui peut envoyer un ou plusieurs messages :***FORMAT*** aux objets contenus dans ces paramètres.

Ainsi en partant de :

```
(1 2) 3  + (4 5) 6
```

qui est en réalité réalisée par trois applications scalaires de la fonction + , sur les couples (1 4), (2 5) et (3 6), on peut conceptualiser le fonctionnement des fonctions scalaires sur les objets en disant que la fonction plus est réellement définie par :

```
+ ← :PLUS⍎
```

le message **:*PLUS*** étant le message primitif envoyé aux scalaires des opérandes de +.

La même démarche est possible pour les autres fonctions scalaires et pour les fonctions de restructuration ou mixtes. En fait l'idée est que le système envoie implicitement des équivalents fonctionnels lors de l'application d'une fonction primitive à un objet tels que :

```
+ ↔ :ADD⍎
− ↔ :SUB⍎
× ↔ :MULT⍎
! ↔ :FACT⍎
⍕ ↔ :FORMAT
```

Cela implique d'avoir des noms de messages reservés. Dès lors une classe est libre d'implanter ou non une ou plusieurs de ces méthodes.

Dans cette optique, il n'y a dès lors plus d'impossibilité à envoyer un message à un ensemble (éventuellement hétérogène) d'objets. L'implantation de cette extension n'est pas prévue dans l'immédiat, mais son intérêt évident - tant théorique que pratique- en fait un candidat de choix pour les travaux ultérieurs.

Une description complète de la syntaxe sous forme BNF que nous proposons se trouve en annexe2. L'idée est qu'un sélecteur de message, soit de même classe qu'une fonction primitive ou définie.

2.3 LES CLASSES DANS OBJAPL 90

Une classe est une zone de travail contenant quelques variables du système de nature particulière :

- ⎕*INL* ("instance variables name list"). C'est un vecteur ou une matrice de caractères qui contient les noms des variables qui seront locales à chaque instance (ce sont des variables d'instance). Toute variable définie dans la zone et qui n'apparait pas dans cette liste sera considerée comme variable de classe (variable dont la valeur sera la même pour toutes les instances de la classe).

- ⎕*MNL* et ⎕*DNL* qui sont respectivement les listes des méthodes monadiques et diadiques. Ces variables sont des matrices ou des vecteurs caractères. Chaque ligne de la matrice contient un nom de sélecteur de message suivi d'une expression APL quelconque décrivant la méthode correspondance à appliquer pour tous les objets qui sont instances de la classe. Dans de telles expressions, ⍵ représentera l'objet courant, et ⍺ le paramètre éventuel du message. L'expression peut comporter plusieurs instructions APL séparer par

un ⋄. Le résultat de la méthode est celui de la dernière instruction.
La transmission de message apparaît comme une opération simple : quand un message est envoyé à un objet, son sélecteur est recherché dans la table appropriée; le système assigne à ⍵ un pointeur sur l'objet courant, à ⍺ le paramètre, et exécute le corps de la méthode dans le contexte de la classe de l'objet.

Cependant, ceci n'est pas suffisant pour traiter toutes les interactions avec un objet. Les variables `⎕MNL` et `⎕DNL` fournissent des réponses pour les messages explicites comme `:CLOSE`, ou des messages semi-explicit tels que `:ADD` envoyé par la fonction APL +. Une classe doit aussi prendre en charge des messages implicites, envoyés par le système. Ces messages sont nécessaires pour :

- initialiser une instance à sa création. Cette opération a lieu juste après l'allocation des structures de données de l'instance.
- détruire une instance. Ce message est envoyé lorsqu'il n'y a plus de référence à l'objet, juste avant la libération de ses structures de données.
- visualiser un objet, en cas d'exécution d'une expression telle que :

```
⎕←OBJET
```

ou

```
OBJET
```

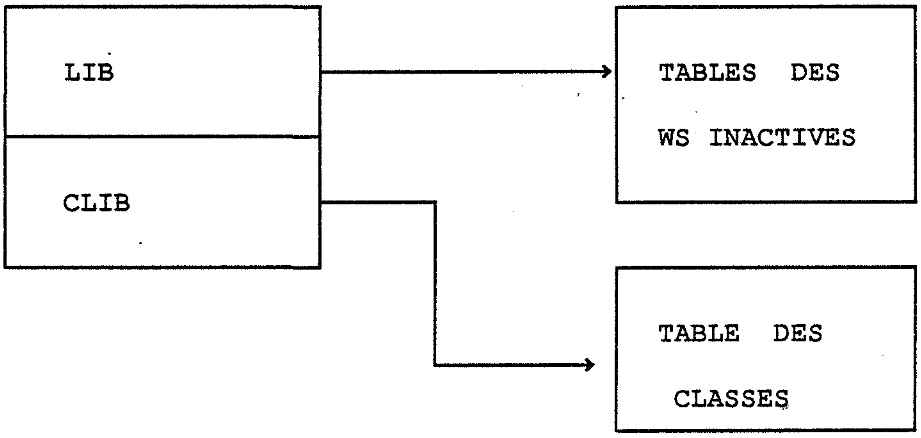
Les méthodes correspondantes sont appelées *méthodes spéciales* et seront définies dans la variable système `⎕SNL` ("special methods name list") qui respecte les mêmes spécifications que `MNL` et `DNL`.

Une classe donc peut être représentée par une table de symboles contenant les références aux méthodes, aux variables de classes et aux variables d'instance.

Du point de vue de la réalisation une classe n'est rien d'autre qu'une WS inactive. Pour des problèmes de cohérence, de même que les WS inactives sont attachées à l'espace virtuel, les classes seront également attachées à l'espace virtuel du système APL (plus précisement elles seront attachées à un espace virtuel particulier car APL 90 sait gérer plusieurs espaces virtuels).

Dès lors de même qu'il existe la commande système `LIB` qui fournit la liste des zones inactives attachées à un espace virtuel, la commande `CLIB` est implantée pour donner la liste des classes attachées à l'espace virtuel. La syntaxe est la suivante :

```
)CLIB
```



La création d'une classe nécessite l'usage des mêmes mécanismes standards que les WS d'APL. Créer une classe revient donc à éditer une zone active vierge, la nommer, y travailler (création de méthodes, de variables d'instance ou de classe ...) et la sauvegarder. Pour distinguer une classe d'une zone de travail traditionnelle, nous utilisons une option spécifique dans la commande du système :save

```
)SAVE -CLIB {NOM_CLASSE}
```

Dès lors le système compile la description de la classe, et génère les structures internes appropriées. Ces structures sont décrites dans le chapitre 6.

L'environnement opérationnel d'une classe est similaire à celui d'une zone de travail. En particulier les variables du système comme ⎕*IO*, ⎕*PP* ou ⎕*CT* sont locales à la classe, et leurs valeurs dans la zone active ne sont pas utilisées.

Ce qui renforce l'idée d'encapsulation et d'entité autonome pour une classe, et est surtout compatible avec le comportement actuel du système APL , qui utilise les valeurs des variables du système initialisés par l'utilisateur uniquement dans les fonctions dont le résultat dépend de ces valeurs. C'est par exemple le cas de ι et de = dont le résultat dépend respectivement de ⎕*IO* et de *CT*.

En particulier, le fait d'initialiser ⎕*CT* de manière que `23=26` rend `1` comme résultat, n'implique pas que *V*`[23]` et *V*`[26]` rendent le même résultat. Pour la même raison si ⎕*IO* est mal défini (tel que ⎕*IO*←'*HELLO*'), le système sait traiter `+/`*V* par accès successif aux éléments de *V*, même si *V*`[`*I*`]` doit provoquer une erreur pour ⎕*IO*.

2.4 INSTANCIATION DANS OBJAPL 90

L'instanciation consiste à créer un objet selon le modèle de sa classe. Nous avons introduit une nouvelle fonction du système ⎕*NEW* sous sa forme diadique.

La syntaxe de l'instanciation est :

`<PARAMETRES>    ⎕NEW {NOM_CLASSE}`

`⎕NEW` crée un nouvel objet (ou instance) de la classe dont le nom figure en argument droit, et lui envoie le message

`<PARAMETRES>  :INIT`

c'est à dire que l'argument gauche de `⎕NEW` est transmis comme paramètre au message implicite `:INIT`. l'argument gauche peut contenir les valeurs locales des variables d'instance.

L'argument droit de `⎕NEW` peut être indifféremment un vecteur de type caractère ou un atome. Un atome est un type de donnée spécifique d'APL 90 qui correspond à la représentation interne des identificateurs. Les atomes sont crées par l'utilisation de la notation entre accollades (voir [Girardot85] au sujet de ces notions).

2.5 HERITAGE DANS OBJAPL 90

La puissance d'APL nous permet de manipuler des objets très complexes (fichiers, bases de données) qui sont relativement indépendants les uns des autres. L'héritage dans APL pose des problèmes qui sont plus d'ordre philosophique que de réalisation. Par ailleurs nous manquons d'expérience de programmation orientée objet pour choisir dans le contexte d'APL entre héritage simple ou héritage multiple. C'est la raison pour laquelle l'implémentation de l'héritage ne nous parait pas indispensable dans une première phase. Le postulat P5 qui dit qu'il existe une hiéharchie entre les classes n'est pas respecté dans notre modèle.

Néanmoins, nous avons choisi nos structures de données d'objet et de classe de manière à prendre en compte facilement l'héritage simple. Quand à l'héritage multiple, nous exposons dans le chapitre 6 la vision selon laquelle elle peut être introduite dans OBJAPL 90.

CHAPITRE 5

Chapitre 5

LES OBJETS DANS L'UNIVERS APL

1 INTRODUCTION

Ce chapitre sert à illustrer l'usage de l'extension objet au delà des concepts. Dans un premier temps nous exposerons une petite application que nous résoudrons en APL classique. Dans une deuxième phase, nous démontrerons l'apport de la P.O.O. dans la bonne intégration de cette applications dans l'univers APL.

Nous nous servirons également de cet exemple pour mettre en évidence les problèmes soulevés par la compatibilité avec APL 2, ainsi que les solutions que nous leur apportons.

2 UNE APPLICATION : L'ARITHMETIQUE DES TRES GRANDS ENTIERS

Un des problèmes de l'informatique théorique est la détermination exacte des ressources requises pour effectuer les opérations arithmétiques élémentaires. Les chercheurs ont été intéressés en particulier par la multiplication, la division, la mise au carré, la racine carrée entière dans un contexte où les opérandes peuvent avoir une taille démesurée. Ces entiers sont souvent désignés par le terme "BIGNUMS" (de l'anglais "BIG NUMBERS").

Nous ne nous intéresserons ici ni à la théorie des "BIGNUMS" ni à l'étude de la complexité des différents algorithmes qu'on rencontre. Pour cela, le lecteur pourra consulter [Brassard86] dans lequel en plus des éléments théoriques, il trouvera de précieuses références bibliographiques.

Nous nous intéresserons à la faisabilité de l'arithmétique des grands entiers par la réalisation de quelques fonctions définies en APL qui implantent les opérations telles que l'addition, la multiplication, les comparaisons etc. Nous n'avons pas cherché à implanter les algorithmes les plus performants. Comme certains systèmes (LE_Lisp, [Chailloux85] nous proposons à l'utilisateur des outils pour traiter les "BIGNUMS".

Nous nous servirons d'une session APL 90 pour illustrer notre application que nous écrirons en premier lieu dans un environnement APL classique, en second lieu, dans un contexte orienté objet.

Nous partirons d'une WS vierge

```
      )CLEAR
CLEAR WS   86/10/04  08.46.21
```

Les bignums sont représentés comme une liste de nombre représentant la valeur du bignum dans une base quelconque. Par exemple en LISP, le bignum qui s'imprime comme 1234567890 peut être représenté comme une liste de la forme (890 567 234 1) par rapport à la base 1000. Nous nous limiterons en ce qui nous concerne à des nombres positifs ou nuls en base 10.

L'idée de ranger les bignums à l'envers repose sur la constation que dans ce type de représentation l'ordre d'affectation des puissances de 10 est le même que les indices (APL) des chiffres de la représentation. Ce qui facilite leur manipulation.

En APL un bignum sera représenté par un vecteur numérique de longueur arbitraire, contenant les valeurs successives des chiffres de la représentation en base 10 de ce nombre.

Définissons quelques fonctions :

- Addition de deux bignums

```
    ∇ Z←X BIGADD Y;N
[1]  LP: N←⌈/(ρX),ρY
[2]   Z←(N↑X)+N↑Y
[3]   →(∧/Z<10)/0
[4]   X←10|Z
[5]   Y←0,⌊Z÷10
[6]   →LP
    ∇
```

- Edition d'un bignum

```
    ∇ Z←BIGEDIT N
[1]   Z←⌽'0123456789'[N]
    ∇
```

- Egalité de deux bignums

```
    ∇ Z←X BIGEGAL Y;N
[1]   N←⌈/(ρX),ρY
[2]   Z←(N↑X)∧.=N↑Y
    ∇
```

- Multiplication de deux bignums

```
    ∇ Z←X BIGMULT Y
[1]   Z←,0
[2]   →L1
[3]  LP: Y←1↓Y
[4]   →((ρY)=0)/0
[5]   X←0,X
[6]  L1: Z←Z BIGADD X×Y[0]
[7]   →LP
    ∇
```

- Normalisation d'un bignum

```
    ∇ Z←BIGNORM X
[1]   Z←((1-ρX)⌈-+/∧\0=⌽X)↓X
    ∇
```

- Soustraction de deux bignums

```
    ∇ Z←X BIGSUB Y;N
[1]  LP: N←⌈/(ρX),ρY
[2]   Z←(N↑X)-N↑Y
[3]   →(∧/Z≥0)/ND
[4]   →(0>¯1↑Z)/ERR
[5]   X←Z BIGADD 10×Z<0
[6]   Y←0,Z<0
[7]   →LP
[8]  ND: Z←BIGNORM Z
[9]   →0
[10] ERR: 'NEG. BIGNUM'
[11]  Z←,0
    ∇
```

Voici quelques exemples d'utilisation :

```
      2538 × 315                         8 3 5 2 BIGMULT 5 1 3
799470                              0 7 4 9 9 7 0 0
```

Nous obtenons le même résultat mis à part les zéros non significatifs qui apparaissent en tête du résultat de BIGMULT. Pour supprimer ces zéros, il suffit de normaliser par l'application de la fonction BIGNORM :

```
      BGNORM 8 3 5 2 BIGMULT 5 1 3
0 7 4 9 9 7
```

```
      V←10000000000000000000000000

      V
1.E25

      V × V
DOMAIN  ERROR
      V × V
        ∧
```

Ce dernier résultat est évidemment dû à un dépassement de capacité. Transformons *V* en BIGNUMS ;

```
  BV ← 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
  ρBV
26
  ρR←     BV BIGMULT BV
51
   R
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

  R BIGADD R
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2

  R BIGADD 9 8 7 6 5 4 3 2 1 9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1 9 8 7 6 5 4 3 2 1 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

  1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 BIGEGAL 1 5 6 3 2 4 8 7 9
0

      R BIGEGAL R
1


      BIGEDIT BV
10000000000000000000000000

      )FNS
BIGADD   BIGEDIT BIGEGAL BIGMULT BIGNORM BIGSUB

      )VARS
BV  R   V
```

Notons au passage que la fonction d'addition peut servir à convertir un nombre en bignum :

```
      0 BIGADD 5315
5 1 3 5
```

Sauvegardons cette WS active dans une WS inactive de nom BIGNUMS_WS.

```
      )WSID
IS CLEAR WS, CLEARED 86/10/04 10.17.35
      )WSID BIGNUMS_WS
WAS CLEAR WS
      )SAVE
BIGNUMS_WS SAVED 86/10/04 10.18.08
      )CLEAR
CLEAR WS   86/10/04  10.19.51
      )FNS
      )VARS
      1 2 3 4 BIGADD 7 8 9 0 7 8 6
VALUE ERROR
      1 2 3 4 BIGADD 7 8 9 0 7 8 6
              ^
```

Pour utiliser les fonctions que nous venons de définir, il faut bien sur les ramener dans la WS active soit par :

```
      )COPY BIGNUMS_WS { LISTE DES ENTITES }
```

Soit par :

```
      )LOAD BIGNUMS_WS
BIGNUMS_WS  SAVED 86/10/04  10.18.08
      )FNS
BIGADD   BIGEGAL BIGMULT BIGNORM BIGEDIT BIGSUB

      )VARS
BV   R   V
```

Les inconvénients de cette technique de programmation sont clairs. Chaque fois que l'on désire traiter les BIGNUMS dans une WS différente de BIGNUMS_WS, il faut ramener les fonctions utiles par l'un des mécanismes déjà décrits.

Un usage correct de ces programmes nécessite en outre, la connaissance de chaque fonction pour comprendre la structure de ses paramètres d'appel. Dans une WS toutes les fonctions définies cohabitent, ainsi que toutes les variables globales. Il n'y a aucune structuration, aucun lien a priori entre telle variable et telle fonction. L'utilisateur éventuel est à la merci de la moindre erreur de manipulation, telle que l'appel d'une fonction avec des arguments qui ne sont pas des bignums...

```
      BIGEDIT 2 3 5 BIGADD ¯3555
INDEX ERROR
BIGEDIT[1]Z←⌽'0123456789'[N]
                ∧
```

Il n'y a aucune encapsulation. Sans une bonne documentation de préférence intégrée, l'exploitation de cette zone de travail est très pénible.

Définissons une fonction qui calcule la factorielle d'un nombre.

```
      ∇ R←FACT N
[1]    →0×⍳N=R←1
[2]    R←N×FACT N-1
      ∇

      FACT 5
120

      FACT 20
2.432902008E18

      FACT 35
DOMAIN ERROR
      FACT 35
      ∧
      FACT 100
DEPTH ERROR
```

Le calcul de la factorielle de 35 n'est pas possible à cause de la limitation de la représentation des nombres sur la machine, le calcul de la factorielle de 100 a avorté à cause de la "sécurité" sur la taille de la pile d'exécution. Nous pouvons augmenter la taille de la pile par la commande système

```
      )DEPTH N
```

et relancer l'exécution. Nous tomberons alors sur résultat analogue à celui de

```
      FACT 35
```

Passons 100 en BIGNUMS :

```
      N ← 0 0 1
      FACT N
DOMAIN ERROR
FACT[1]→0×⍳N=R←1
              ∧
```

Il y a cette fois échec à l'exécution parce que la fonction = que nous utilisons ne travaille pas correctement sur les BIGNUMS. Il en aura été de même pour la soustraction et la multiplication. Pour résoudre ce problème, il faut écrire une fonction de calcul de la factorielle spécifique aux BIGNUMS, ce qui revient à utiliser les méthodes BIGEGAL, BIGSUB et BIGMULT à la place de =, − et ×.

La fonction factorielle propre aux BIGNUMS s 'écrit :

```
    ∇ R←BIGFACT N
[1]  →0×⍳N BIGEGAL R←1
[2]  R←N BIGMULT BIGFACT N BIGSUB 1
    ∇

      BIGFACT 5 3
0 0 0 0 0 0 0 2 3 2 5 7 3 3 1 5 6 6 6 6 9 2 9 4 4 1 6 8 3 6 6 9 7
      4 1 3 3 3 0 1
```

Ce processus sera nécessaire à chaque fois que nous voudrons traiter des objets d'un autre type. Moralité, les bignums ne sont pas des "first class citizens".

3 LA MANIPULATION DES OBJETS DANS OBJAPL

Nous allons maintenant traiter l'application précédente dans un contexte orienté objet. Pour cela nous allons créer une classe des BIGNUMS et définir des méthodes auxquelles ces BIGNUMS peuvent répondre.

3.1 CREATION D'UNE CLASSE

Créons une zone de travail de nom BIGNUMS_CLASSE :

```
      )CLEAR
CLEAR WS   86/10/04  10.29.51
      )WSID BIGNUMS_CLASSE
WAS CLEAR WS
      )FNS
      )VARS
```

Pour que cette WS devienne une classe il faut la sauvegarder dans la bibliothèque des classes par la commande suivante :

```
      )SAVE -CLIB BIGNUMS_CLASSE
BIGNUMS_CLASSE SAVED 86/10/04 10.28.08
```

Si on omettait l'option -CLIB, BIGNUMS_CLASSE serait considérée comme une WS inactive classique.

Toute variable globale définie dans cette classe est considérée comme une variable de classe dont la valeur sera identique pour toutes les instances de la classe.

Les variables d'instance seront, déclarées par l'affectation de la variable du système ⎕*INV*

Ainsi la séquence

```
BIGN← 0
⎕INV ← { BIGN }
```

fait de ***BIGN*** une variable d'instance dont la valeur par défaut est `0`.

La différence entre variables de classe et variables d'instance se note par leur appartenance ou non à ⎕***INV***. Dès lors on notera qu'il devient aisé de modifier les champs d'une classe.

La classe BIGNUMS que nous venons de définir admet deux variables de classe de noms ***A*** et ***B*** et une seule variable d'instance de nom BIGN et dont la valeur est 0 par défaut.

sauvegardons le nouvel état de la classe :

```
      )SAVE -CLIB
BIGNUMS_CLASSE SAVED 86/10/04 10.28.08
```

3.2 CREATION D'INSTANCE

Créons deux instances de BIGNUMS de noms respectifs A et B en partant d'une WS vierge :

```
      )CLEAR
CLEAR WS    86/10/04  10.29.51
      )FNS
      )VARS
      A ← 8 3 5 2  ⎕NEW {BIGNUMS_CLASSE}
      B ← 3 1 5  ⎕NEW {BIGNUMS_CLASSE}
```

A et B sont définis comme deux objets de la classe BIGNUMS qui admettent comme variable d'instance respectivement le vecteur `8 3 5 2` et `3 1 5`.

3.3 DEFINITION DES METHODES D'INSTANCE

Jusqu'ici, nous avons défini une classe BIGNUMS et deux de ses instances, A et B, dont les états sont respectivement :

`2 5 3 8` et `3 1 5.`

Ces objets ne peuvent répondre pour l'instant à aucun message spécifique car leur classe ne contient aucune description de comportement. Dans Smalltalk, on dispose de toute une panoplie de messages permettant de rajouter, de supprimer ou de modifier les méthodes ou les variables d'une classe.

Pour des raisons pratiques et surtout de mise au point, il faut un mécanisme conversationnel pour éditer aussi bien les méthodes que les variables.

Dans un contexte APL, il nous semble plus harmonieux et plus efficace de proposer un véritable mode d'édition d'une classe. Une classe étant très semblable à une zone de travail, la commande *LOAD* est étendue :

```
      )LOAD -CLIB NOM_CLASSE
```

Continuons notre session :

```
      )WSID TEST_WS
WAS CLEAR WS
      )FNS
      )VARS
A   B
      )SAVE
TEST_WS SAVED 86/10/04 10.28.08
      )CLEAR
CLEAR WS   86/10/04  10.29.51
```

Pour éditer notre classe BIGNUMS_CLASSE on va la charger :

```
      )LOAD -CLIB BIGNUMS_CLASSE
BIGNUMS_CLASSE SAVED 86/10/04 10.38.08
      )VARS
BIGN
      BIGN
0
```

Pour définir les méthodes, on utilise les techniques classiques de la définition de fonction en APL. L'idée est de traduire les fonctions définies :

```
BIGADD BIGEDIT BIGEGAL BIGMULT BIGNORM BIGSUB
```

en méthodes de la classe BIGNUMS. Ainsi si BIG1 et BIG2 sont deux instances de BIGNUMS_CLASSE, leur somme s'ecrirait :

```
BG1 :BIGADD  BG2
```

Cependant ces fonctions ne sont plus utilisables telles quelles. En effet les paramètres formels qu'elles utilisent sont des variables ordinaires APL et non des identificateurs d'objets définis. En fait ces variables formelles telles que définies ci dessous représentent les valeurs des variables d'instances des objets qui eux n'apparaissent pas explicitement.

D'une part il faut pouvoir établir un lien entre la variable d'instance BIGN et les paramètres formels manipulés par ces fonctions. D'autre part il est nécessaire d'obtenir la valeur de la variable d'instance BIGN locale à l'objet BG1 dans l'appel :

```
BG1 :BIGADD  BG2
```

En effet quand on exécute BIGADD, on est dans le contexte de BIG2, donc BIGN a une valeur. Pour obtenir la valeur de BIGN reltive à BG2, nous proposons par exemple la méthode :

```
:BIGN BG2
```

qui signifierait "rendre la valeur de BIGN", soit :

```
    ∇ Z← :BIGN ω
[1] Z← BIGN
    ∇
```

Remarquons que pour l'instance courante (l'objet dans le contexte duquel on se trouve) les valeurs des variables d'instance s'obtiennent uniquement en spécifant leurs noms (comme dans une WS APL)

Dès lors une version correcte (sous forme α+ω) des méthodes de la classe BIGNUMS serait :

- méthodes monadiques :

```
BIGEDIT :  ⌽'0123456789'[BIGN]
BIGN  :   BIGN
```

- méthodes diadiques :

```
BIGADD :     (NORM (:BIGN α) ADD BIGN) ⎕NEW {BIGNUM}
BIGEGAL :   (:BIGN α) EQUAL BIGN
BIGMULT :    (NORM (:BIGN α) MULT BIGN) ⎕NEW {BIGNUM}
BIGSUB :     (NORM (:BIGN α) SUB BIGN) ⎕NEW {BIGNUM}
```

- méthodes Speciales :

```
INIT :   BIGN←,α ◇ ω
PRINT :  ⎕←:FORMAT ω ◇
```

On voit apparaître un problème : certaines méthodes (BIGADD, BIGSUB,...) doivent rendre impérativement comme résultat un objet de même classe (BIGNUMS) alors qu'il est souhaitable que d'autres comme (BIGEGAL, BIGEDIT...) rendent comme résultat des booléens ou des vecteurs au sens classique de ces termes. En arrière plan de cette constation se trouve en fait les difficultés liées aux phénomènes de conversion de types de manière générale.

Les fonctions ADD, EGAL, MULT, NORM, EDIT et SUB sont définies dans la classe BIGNUMS_CLASSE comme ci-dessous:

```
     ∇ Z←X ADD Y;N
[1]   LP: N←⌈/(ρX),ρY
[2]    Z←(N↑X)+N↑Y
[3]    →(∧/Z<10)/0
[4]    X←10|Z
[5]    Y←0,⌊Z÷10
[6]    →LP
     ∇
     ∇ Z←EDIT N
[1]    Z←⌽'0123456789'[N]
     ∇
     ∇ Z←X EGAL Y;N
[1]    N←⌈/(ρX),ρY
[2]    Z←(N↑X)∧.=N↑Y
     ∇
     ∇ Z←X MULT Y
[1]    Z←,0
[2]    →L1
[3]   LP: Y←1↓Y
[4]    →((ρY)=0)/0
[5]    X←0,X
[6]   L1: Z←Z ADD X×Y[0]
[7]    →LP
     ∇
     ∇ Z←NORM X
[1]    Z←((1-ρX)⌈-+/∧\0=⌽X)↓X
     ∇
     ∇ Z←X SUB Y;N
[1]   LP: N←⌈/(ρX),ρY
[2]    Z←(N↑X)-N↑Y
[3]    →(∧/Z≥0)/ND
[4]    →(0>¯1↑Z)/ERR
[5]    X←Z ADD 10×Z<0
[6]    Y←0,Z<0
[7]    →LP
[8]   ND: Z←NORM Z
[9]    →0
[10] ERR: 'NEG. BIGNUM'
[11]   Z←,0
     ∇
```

Nous avons terminé la définition des méthodes. Si nous sauvegardons cette classe dans son état actuel, aucune de ces méthodes ne sera accessible de l'extérieur, car notre classe sera conservée sous forme de WS inactive sans point d'entrée. Les points d'entrée permettent de faire le lien entre le nom externe d'une méthode et son nom interne. Les noms externes sont des identificateurs distingués commmençant par le signe **:** les noms internes suivent les règles relatives aux identificateurs en APL.

Ainsi soit :

```
      MM
:BIGEDIT ⍕'0123456789'[BIGN]
:BIGN   BIGN
      ⎕MNL←MM
      DM
:BIGADD     (NORM (:BIGN α) ADD BIGN) ⎕NEW {BIGNUM}
:BIGEGAL   (:BIGN α) EQUAL BIGN
:BIGMULT    (NORM (:BIGN α) MULT BIGN) ⎕NEW {BIGNUM}
:BIGSUB     (NORM (:BIGN α) SUB BIGN) ⎕NEW {BIGNUM}
      ⎕DNL←DM
      SM
:INIT   BIGN←,α ⋄ ω
:KILL
:PRINT  ⎕←:FORMAT ω ⋄
      ⎕SNL←SM
```

définissent les points d'entrées monadiques, diadiques et spéciales de la classe.

Ce choix de ne pas rendre tout accessible peut sembler curieux à première vue. Il se justifie par le fait que dans la classe peut exister un certain nombre de fonctions de services utiles uniquement au sein des méthodes de la classe, qui n'ont donc pas à être accédées de l'extérieur. Ces fonctions n'ont de sens que dans le contexte de la classe BIGNUMS_CLASSE, et ne doivent jamais être connues à l'extérieur. C'est le cas de ADD, MULT, etc... Ceci renforce l'encapsulation des données et des procédures dans la mesure où on peut dire qu'il existe deux types de méthodes : les unes publiques (les points d'entrée) les autres privées (les utilitaires de service).

Par ailleurs la dissociation nom externe et nom interne permet de définir plusieurs points d'entrée pour la même méthode interne. Les trois variables que nous venons de définir sont conservées par le système pour réexploration en cas de modification de la classe.

```
      )WSID
IS CLASS: BIGNUM_CLASSE
      )FNS
BIGADD BIGEDIT BIGEGAL BIGMULT BIGNORM BIGSUB
ADD EDIT EGAL MULT NORM SUB
      )VARS
BIGN DM MM SM
```

Sauvegardons la classe

```
      )SAVE -CLIB
BIGNUMS_CLASSE SAVED 86/10/04 10.39.08

      )CLEAR
CLEAR WS   86/10/04  10.39.51

      C← 2 3 5 6 7 8 9 ⎕NEW {BIGNUMS_CLASSE}

      D← 3 5 8  ⎕NEW {BIGNUMS_CLASSE}

      )VARS
C  D

      )FNS
```

Il n'existe aucune fonction définie

```
      C :BIGADD D
5 8 3 7 7 8 9 0

      C :BIGMULT D
6 9 7 1 8 6 4 2 4 8 0 0

      T← ρι6

      M←'QWERTY'

      )VARS
C D T M
      T BIGMULT T
VALUE ERROR
      T BIGMULT T
        ∧
```

En effet ***T*** n'est pas un objet défini (il n'est pas instance d'une classe), la recherche de l'identificateur ***BIGMULT*** a été faite dans la table de symboles de la WS active et comme il n'existe aucune fonction définie de ce nom, on génère une erreur.

```
      M :BIGEGAL M
NOT AN OBJECT
      M :BIGEGAL M
                 ∧
      C :BIGSUP D
MSG NOT FOUND
      C :BIGSUP D
        ∧
```

Dans le premier cas, l'utilisation d'un sélecteur nous indique que la recherche de méthodes doit se faire dans la table de symboles de l'objet ***M***. Malheusement ***M*** n'est pas un objet défini.

Dans le second cas *D* et *C* sont bien des objets de même classe (BIGNUMS_CLASSE), mais il n'existe pour l'instant aucune méthode correspondant à l'identificateur BIGSUP. Plus précisement :BIGSUP n'est pas un point d'entrée de la classe BIGNUMS_CLASSE.

```
      )LOAD TEST_WS
TEST_WS SAVED 86/10/04 10.28.08
      )FNS
      )VARS
A   B

      A :BIGADD B
5 6 8 8
```

A partir de maintenant, on s'aperçoit que les BIGNUMS peuvent être définis dans tout environnement actif ou inactif. Les messages auxquels ils répondent sont factorisés dans la classe BIGNUMS de manière transparente à l'utilisateur.

3.4 LA MODIFICATION DES CHAMPS OU DES METHODES

APL dispose de ces propes techniques traditionnelles, de manipulation des données, et, dès que l'on active une classe, l'éditeur de fonction APL reste disponible. Une méthode n'étant rien d'autre qu'une fonction définie APL, toutes les manipulations qu'on peut faire sur les fonctions définies, sont également possibles sur les méthodes. Par exemple :

```
      )LOAD -CLIB BIGNUMS_CLASSE
BIGNUMS_CLASSE SAVED 86/10/04 10.50.08
      )VARS
BIGN
      )FNS
BIGADD  BIGEDIT BIGEGAL BIGMULT BIGNORM BIGSUB
ADD EDIT EGAL MULT NORM SUB
```

ce qui représente la liste des méthodes et des fonctions internes de la classe.

```
      )ERASE LISTE_DE_NOMS
```

pour détruire des méthodes ou des variables de classe.

```
      ∇ NOM_METHODE[⎕]∇
[1]           .
[2]           .
[3]           .
[4]           .
              .
              .
      ∇
```

pour afficher le corps de la méthode ou la modifier

```
      ∇ NOM_METHODE
[1]           .
[2]           .
[3]           .
[4]           .
               .
               .
      ∇
```

pour rajouter une nouvelle méthode.

Il est par ailleurs utile d'avoir dans chaque classe une méthode :DESCRIBE qui donne des informations sur la classe, offrant ainsi une documentation en "ligne". Une variable du système sera attachées à chaque WS :

- ⎕*CLASSE* qui donne le nom de la classe courante

Pour la classe BIGNUMS on peut ainsi avoir :

```
       ∇ DESCRIBE
[ 1]⍝ NOM DE LA CLASSE : {BIGNUMS_CLASSE}
[ 2]' CLASSE DES ENTIERS EN PRECISION INFINIE '
[ 3]' UNE SEULE VARIABLE D''INSTANCE DE NOM BIGN '
[ 4]' VALEUR PAR DEFAULT DE BIGN : ', ⍕ BIGN
[ 5]' LA VALEUR DE BIGN EST SAISIE COMME UN VECTEUR, '
[ 6]' DE CHIFFRE A L''ENVERS '
[ 7]' EXEMPLE : 123456 ↔ 6 5 4 3 2 1 '
[ 8]' CETTE CLASSE NE TRAITE QUE DES ENTIERS ≥0 '
[ 9]' VOICI LES METHODES MONADIQUES RECONNUES '
[10]⎕MNL
[11]' VOICI LES METHODES DIADIQUES RECONNUES '
[12]⎕DNL
[13]' VOICI LES METHODES SPECIALES RECONNUES '
[14]⎕SMNL
       ∇
```

Cette méthode doit être dans ⎕*SNL* et on peut obtenir son exécution depuis n'importe quel espace de travail par la commande système :

```
)DESCRIBE {NOM_CLASSE}
```

Notons que cela n'empêche nullement d'avoir une méthode :DESCRIBE qui cette fois sera dans ⎕*MNL* et décrit un objet. Il suffit que les deux points d'entrée ne fassent pas appel à la même fonction interne.

Un bignum particulier répondra au message :DESCRIBE en faisant appel à la fonction interne :

```
    ∇ INFO ⍺
[1]   ⍝ INFOS SUR UNE INSTANCE DE BIGNUMS_CLASSE
[2]   ' LA VARIABLE D''INSTANCE BIGN A POUR VALEUR '
[3]   BIGN
[4]   ' LE NOM DE SA CLASSE EST : '
[5]   ⎕CLASSE
    ∇
```

3.5 LES CONVERSIONS DE TYPES DE DONNEES

Les mécanismes de conversion de types de données est un aspect fondamental d'APL. Le choix d'APL est de traiter les booléens, les entiers, et les réels comme des types dérivés du concept abstrait de "nombre", le programmeur restant (théoriquement) dans l'ignorance de la forme de stockage de ses données. Comment maintenir cette souplesse dans un contexte orienté objet où les nouveaux types sont imprévisibles ? Quels mécanismes de coercition doit on définir entre les types primitifs et les types définis par l'utilisateur ? Et plus particulièrement de quelle manière le programmeur peut-il indiquer non seulement les conversions permises sur les objets, mais aussi les circonstances dans lesquelles ces coercitions doivent avoir lieu ?

Sans réponse effective et efficace à ces questions, toute réalisation de système APL offrant un mécanisme d'abstraction de types de donnée, détériorera aussi bien le style que les techniques de programmation qui caractérisent le langage APL.

Par exemple la fonction de calcul de la factorielle d'un nombre est :

```
    ∇ R←FACT N
[1]   →0×⍳N=R←1
[2]   R←N×FACT N-1
    ∇
```

Pour un bignum nous avons écrit :

```
    ∇ R←BIGFACT N
[1]   →0×ιN BIGEGAL R←1
[2]   R←N BIGMULT BIGFACT N BIGSUB 1
    ∇
```

Depuis que les fonctions ***BIGEGAL BIGMULT BIGSUB*** sont devenus des points d'entrée de la classe BIGNUMS, cette version est devenue caduque. En effet ces fonctions opèrent sur des instances de la classe BIGNUMS et non pas sur des tableaux primitifs. Or dans la ligne 1 de BIGFACT, la variable N désigne un objet de type bignum, ce qui n'est pas le cas de R, qui est le scalaire primitif 1, de type entier ou booléen. La situation est la même pour l'argument droit de BIGSUB dans la ligne 2.

On peut résoudre le problème à la main en corrigeant BIGFACT de la manière suivante :

```
    ∇ R←BIGFACT N
[1]   →0×ιN :BIGEGAL R← 1 ⎕NEW {BIGNUMS_CLASSE}
[2]   R←N :BIGMULT BIGFACT N :BIGSUB  1 ⎕NEW {BIGNUMS_CLASSE}
    ∇
```

Est-il possible d'offrir un mécanisme général de conversion qui soit transparent à l'utilisateur.

La solution adoptée dans les systèmes APL classiques consiste à construire une table, contenant toutes les combinaisons possibles de types, ainsi que les actions à entreprendre dans chaque cas. Ainsi, si A et B sont deux données APL, et F une opération arithmétique l'expression A F B, est traitée en utilisant une table du style :

cas	A	B	⟶	A	B
1	reel	reel		reel	reel
2	entier	reel		reel	reel
3	entier	entier		entier	entier
4	reel	entier		reel	reel

Les colonnes situées à droite de la flèche indiquent les conversions éventuelles à effectuer. Il est absurde de vouloir modifier cette table à chaque fois qu'on crée un nouveau type de donnée, car on retomberait dans un système figé où toute création d'objet necessiterait la modification et la regénération du système. Par aillleurs plus il y aura de nouveaux types plus la complexitée de la table de coercition augmentera.

Cette technique suppose en outre l'existence d'une véritable hiérarchie entre les types de données numériques booléens, entiers, réels et complexes. Ces mécanismes s'appliquent non seulement lorsque les types des données qui interviennent dans une même opération sont différents mais aussi en cas d'evènement exceptionnel comme les problèmes d'"overflow" dans le traitement des entiers.

L'introduction des objets soulève un nouveau problème, il n'est pas aisé de définir une hiérarchie entre tous les objets dans la mesure où on peut en créer de nouveaux à chaque instant. De même en cas de dépassement de capacité dans le traitement des entiers doit-on convertir les opérandes en réels ou en bignums ?

D'autre part ces mécanismes doivent-elles s'appliquer à tous les objets d'un tableau de manière uniforme ou uniquement sur les éléments qui le nécessitent, créant ainsi des résultats hétérogènes à partir de tableaux homogènes. Par exemple dans le résultat de :

```
! 2 3 3245
```

les deux premiers éléments sont des entiers, le dernier élément n'est représentable que sous forme de bignum (avec environ 9500 chiffres...). A l'heure actuelle nous n'avons pas de mécanisme général transparent à l'utilisateur. Les paramètres des messages doivent être convertis de manière explicite par l'utilisateur. Cependant nous partageons l'approche de [Kajiya83] qui préconise que le système fournisse à l'utilisateur la possibilité de décrire ces propres mécanismes de coercisions en les encapsulant dans les types de donnée.

3.6 LES OBJETS ET LA "STRAND NOTATION"

Rappellons qu'elle se définit de la façon suivante : si A, B, C, D, E sont cinq variables alors

```
A B C D E  ↔ (⊂A) , (⊂B) , (⊂C) , (⊂D) , (⊂E)
```

Les variables A, B, C, D, et E peuvent être de types et de structures différentes. Dans la mesure où dans OBJAPL 90 les objets sont définis comme des nouveaux types de donnée, ils s'intègrent harmonieusement à la notation vectorielle. Ainsi si O est de type objet, on a :

```
      A B C D E O ↔ (⊂A) , (⊂B) , (⊂C) , (⊂D) , (⊂E) , (⊂O)
```

Un objet défini est atomique, sa dimension est vide et son rang est nul. En cela il se comporte de la même façon qu'un scalaire de type numérique ou caractère.

Donnons quelques exemples qui explicitent le comportement des objets.

```
      A ←  123
      B ←  '⎕'
      O ←  9 8 7 6 5 4 3 2 1  ⎕NEW {BIGNUMS_CLASSE}


      ρA                ρB                ρO

      ρρA               ρρB               ρρO
0                 0                0


      V ← A  B  O
      ρV               ρρV
3              1
```

V est un vecteur de trois éléments de types hétérogènes. Le premier élément est le scalaire 123 de type numérique, le deuxième est un scalaire '⎕' de type caractère, le troisième est un scalaire de type BIGNUMS.

```
      V
123   ⎕     123456789
```

Remarquons que la valeur de O qui apparaît dans V, est celle qui apparait quand on demande l'impression de O, ce qui est compatible avec le cas de A et de B.

Désormais, dans la notation vectorielle, une position peut être occupée non seulement par un scalaire numérique, un scalaire ou un vecteur caractère, une expression, mais aussi par un scalaire de type objet défini.

3.7 LES MESSAGES ET LES OPERATEURS

Ici nous traiterons uniquement les opérateurs d'APL2, avec lesquels nous cherchons à être compatibles.

- L'opérateur **each** (¨)

Si G est une fonction monadique (primitive ou non) rappelons l'identité suivante :

```
(G¨ W) [I]↔ ⊂G⊃W[I]
```

Nous assurons cette identité sur les sélecteurs de message.

```
        BIG1 ← 5 0 0 ⎕NEW {BIGNUMS_CLASSE}
        BIG2 ← 5 1 0 ⎕NEW {BIGNUMS_CLASSE}
        BIG1 ← 5 3 0 ⎕NEW {BIGNUMS_CLASSE}

        VBIG ← BIG1 BIG2 BIG3

        :BIGEDIT VBIG
NOT AN OBJECT
        :BIGEDIT VBIG
                 ∧

        :BIGEDIT ¨ VBIG
5  15  35

        ! 1  2  3  4
1  2  6  24
```

Si on définit la fonction **:*BIGFACT*** comme méthode de BIGNUMS_CLASSE alors :

```
        :BIGFACT ¨ VBIG
120  1307674368000 10333147966386144929666651337523200000000
```

L'envoi réparti de message par l'intermédaire de l'opérateur EACH assure une grande richesse et une grande souplesse à notre extension.

Ainsi si nous avons défini une classe "WINDOW", représentant des fenêtres sur un écran planipunctique, classe sur laquelle le message :CLOSE est définie, alors EACH permet de fermer l'ensemble des fenêtres F1 F2 F3 et F4 par :

```
        :CLOSE¨ F1 F2 F3 F4
```

Si la fonction primitive scalaire **!** était autorisée sur les objets et interprété par le système comme l'envoi du message **:*BIGFACT*↧** alors on aurait :

```
      M ← 1  BIG1  2  BIG2  3  BIG3  4

      !¨ M
1  120  4  1307674368000 6 10333147966386144929666
                             6513375232000000 24
```

Ce qui explicite davantage l'importante différence comportementale entre tableaux tableaux d'objets et tableaux de scalaires simples, un message ne s'appliquant qu'à un seul objet, une fonction scalaire s'appliquant à une structure complète.

- L'extension des opérateurs primitifs.

La définition des opérateurs primitifs d'APL tels que **la réduction, la propagation, le produit interne et le produit externe** est étendue pour admettre toute fonction (primitive ou non), donc des messages.

Exemple :

```
      + / 1 2 3 4
10

      :BIGADD / BIG1 BIG2 BIG3
55
```

Si la fonction `+` était implicitement applicable aux objets et les conversions transparentes à l'utilisateur, alors on pourrait apprecier la puissance de l'expression suivante :

```
      + / M ← 1  BIG1  2  BIG2  3  BIG3  4
```

qui fournirait comme résultat `65`.

3.8 LES OBJETS ET LA NOTION DE TYPE ET DE PROTOTYPE

Nous avons vu dans la partie 1 que le type d'un tableau T, est un tableau de même structure dans lequel on remplace chaque scalaire primitif par son élément typique. Tous les nombres admettent le même élément typique qui est le chiffre `0`, l'élément typique caractère est le caractère `' '` (blanc). Dans APL 90 nous avons implanté le symbole `⊤` en monadique comme symbole associé à la fonction type d'un tableau. Notons que le mot type ne nous satisfait pas pour nommer cette opération, c'est pourquoi nous parlerons plutôt d'exemplaire typique du tableau T.

Exemple :

```
      T ← A B C D E  (2×A)

      T
123  ⎕  1 2 3   OBJAPL90   JJ  123456789 226
        4 5 6              SS

      T T
0  ∧  0 0 0   ∧∧∧∧∧∧∧∧   ∧∧   0   0
      0 0 0              ∧∧
```

le symbole ∧ symbolisant le caractère "blanc".

L'introduction des objets définis comme types de donnée nous oblige donc à définir l'élément typique pour chaque classe d'objets. Nous utiliserons le symbole pour représenter par défaut un tel élément dans les exemples ci dessous.

```
      P ← ( 1 2 3)  BIG1  (3 2ρ'JJSSFM')  (2 2ρBIG2)
      P
1 2 3     5     JJ      15 15
                SS      15 15
                FM

      T P
0 0 0     $     ∧∧      $ $
                ∧∧      $ $
                ∧∧
```

Pour être compatible avec APL 2 et pour une manipulation aisée des objets dans l'univers APL :

- Un objet doit avoir un exemplaire typique.
- L'exemplaire typique doit avoir les mêmes propriétés que les autres intances de la classe
- Cet exemplaire typique doit être le même pour toutes les instances d'une même classe.
- Il doit répondre aux mêmes fonctionnalités.

C'est pourquoi nous proposons que l'exemplaire typique d'une classe soit l'instance de la classe qui admet comme valeurs des variables d'instance les valeurs par défaut de ces mêmes variables. Nous parlerons alors **d'instance prototype** d'une classe. L'instance prototype est crée implicitement avec la classe.

Par exemple l'instance prototypique de la classe des bignums sera le bignum 0. Et l'exemplaire typique de P devient

```
      T P
0 0 0    0     ∧∧      0  0
               ∧∧      0  0
               ∧∧
```

Donc à chaque fois qu'il y a création de classe, il y a génération d'une instance prototype[1] qui n'apparait de manière explicite.

Le prototype d'un tableau T étant l'exemplaire typique de son premier élément, cette notion n'impose aucune contrainte particulière pour les objets définis. Rappelons que le prototype s'obtient par l'expression `T↑T`. Ainsi, si O est un objet quelconque, nous pouvons écrire :

```
      Q ← (1 2 O  'APL') (2 3 ρι6)   BIG1

      T Q
0    0    $    ∧∧∧       0 0 0    0
                         0 0 0

      T↑ Q
0    0    $   ∧∧∧
```

Toutes les fonctions de restructuration qui peuvent nécessiter un remplissage complétif, sont dès lors disponibles pour les objets définis. Il suffit d'utiliser le prototype comme élément de remplissage.

```
      ρQ
3

      4 ↑ Q
1   2     O    APL     1 2 3    5     0    0    $   ∧∧∧
                       4 5 6
```

---

[1] Si on étend les fonctions primitives sur les objets, une autre vision consiste à dire qu'il se produit une erreur de domaine si la fonction type T est appliquée à un objet dont la classe n'a pas défini le sélecteur monadique **:** ***TYPE***, contenu dans le message **:** ***TYPE*↓** envoyé par le système à l'objet quand il détecte T. Notre choix actuel a pour but de simplifier la manipulation par l'utilisateur des structures contenant des objets et peut changer ultérieurement.

CHAPITRE 6

0.1

Chapitre 6

QUELQUES STRUCTURES INTERNES DU SYSTEME

1 LES OBJETS ET LA MEMOIRE VIRTUELLE

Certaines particularités internes du système APL 90 ont été favorables à l'extension objet. Notamment :

- la grande taille de l'univers des objets, reconnue habituellement comme une nécessité dans les systèmes orientés objet
- la gestion simultanée de plusieurs tables de symboles (les systèmes traditionnels n'en gère qu'une)
- le changement de contexte d'une zone active à une est très efficace
- l'utilisation de cache pour accéder aux valeurs des symboles est performant même dans les cas où la table des symboles est de très grande taille.

Au niveau de la réalisation, nous voulions satisfaire certaines contraintes :

- prendre appui sur les structures qui existent déjà dans APL 90
- optimiser la transmission de messages à un objet, de manière à ramener son coût à celui d'un appel de fonction.

1.1 DESCRIPTION INTERNE D'UNE CLASSE

Le descripteur d'une classe est un bloc de mémoire contenant les champs suivants :

RU	GDM	ID—Classe		CEC
PT1	PT2	PT3	PT4	PT5
val1	val2	. . . .		valn

RU est une copie de la référence universelle de la classe. C'est une RU à mémoire. Nous avons déjà évoqué le rôle de la RU dans la première partie.

GDM représente un ensemble d'informations relatives à la gestion mémoire (RU et GDM sont présents dans tout bloc de mémoire alloué par le système).

ID-Classe est une identification de classe, elle peut réperer une super classe par exemple dans un contexte d'héritage.

val1, ..., valn représentent les valeurs par défaut des variables d'instance de la classe. CEC est le compteur d'évolution de la classe qui, comparée avec celle des instances, permet de réaliser les mises à jour.

PT1 est le pointeur sur la liste des noms de variables d'instance. C'est le même champ que celui détenu par les instances à jour.

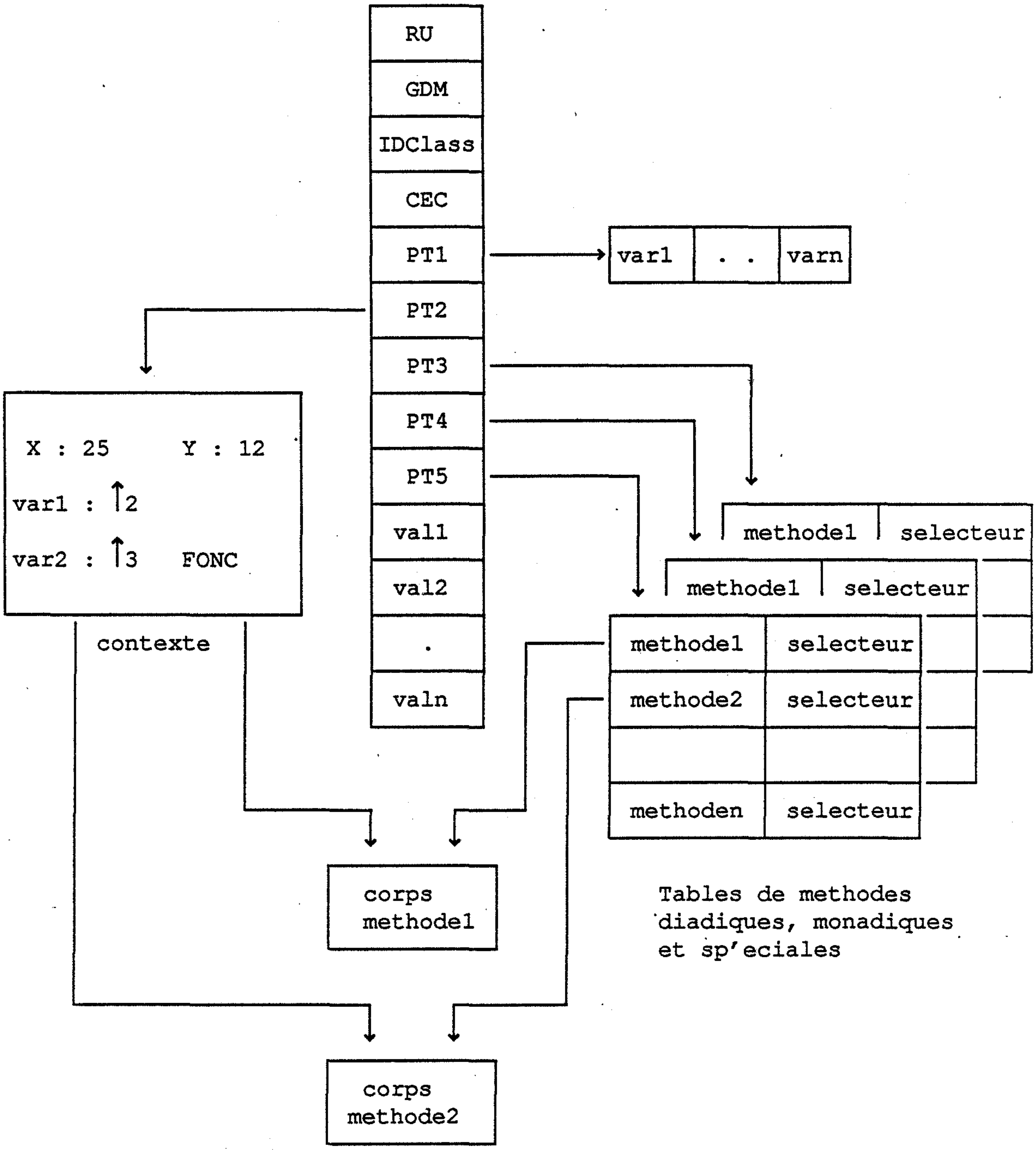
PT2 est l'adresse d'un contexte. C'est celui dans lequel seront exécutées les méthodes attachées à la classe. Ce contexte contient les références et les valeurs des variables de classe, les adresses des corps des méthodes ainsi que les indirections sur les valeurs des variables d'instance. C'est le moyen d'utiliser les valeurs locales des variables d'instances d'un objet.

PT3 est un pointeur sur le dictionnaire des méthodes monadiques de la classe accessibles depuis l'extérieur.

PT4 est un pointeur sur le dictionnaire des méthodes diadiques de la classe accessibles depuis l'extérieur.

PT5 est un pointeur sur le dictionnaire des méthodes spéciales de la classe.

Le schéma ci-dessous donne une vue globale de l'implantation d'une classe.



Les indirections dans le contexte indiquent les entrées dans la table des valeurs des variables, détenues par les instances de la classe.

Lorsqu'au niveau d'une instance, les entrées correspondantes ne sont pas définies, on utilise les valeurs par défaut détenues par la classe de l'objet.

la définition d'une classe implique des tests et des mise à jour. Dans notre modèle, la description d'une classe est compilée à chaque fois qu'on la sauvegarde dans la bibliothèque des classes.

1.2 DESCRIPTION INTERNE D'UNE INSTANCE

Le descripteur d'une instance est simple et diffère peu de celui d'une classe. La structure d'un objet défini est semblable à celle d'un vecteur généralisé. Dans la mémoire virtuelle, un objet (instance d'une classe) sera représenté par un descripteur de type ci-dessous :

RU	GDM	ID—Classe	PT1
CEI	VAL1	. . . . .	VALn

RU et GDM ont la même spécification que pour une classe.

ID-CLASSE est l'identificateur de la classe dont l'objet est instance.

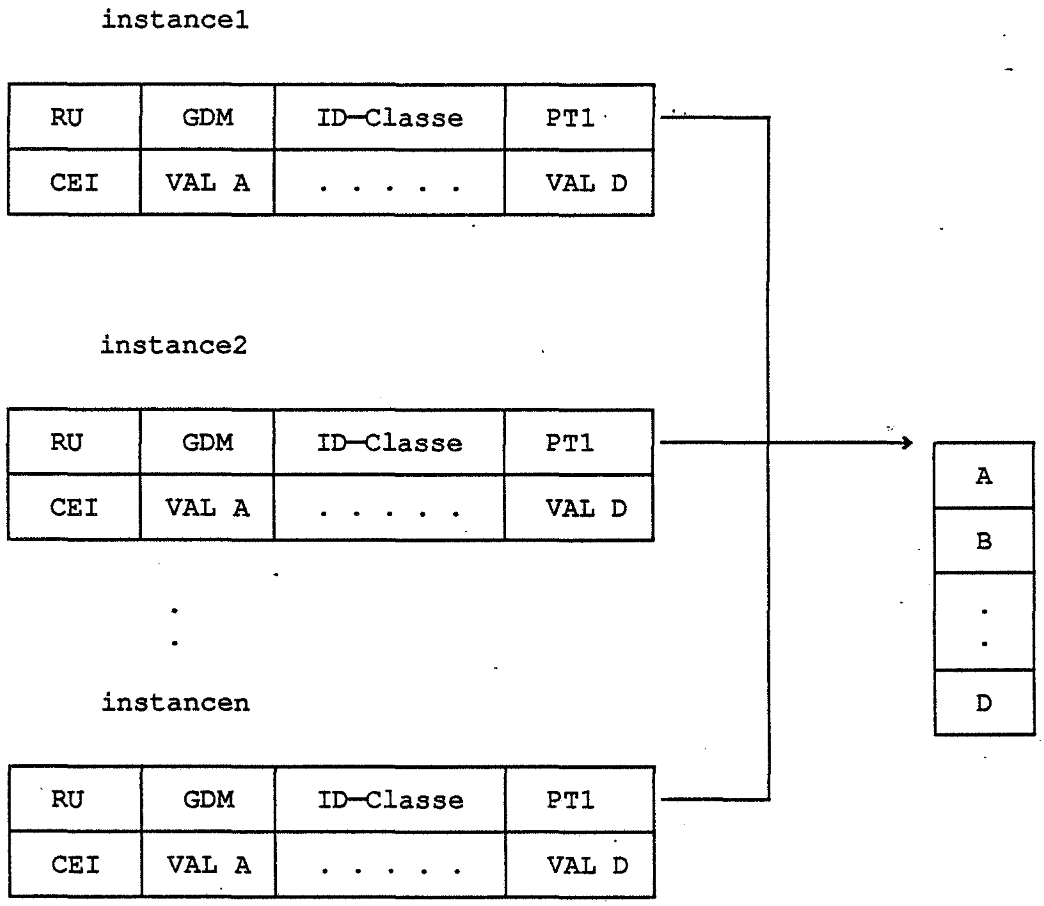
VAL1, VAL2, ..., VALn sont les valeurs des variables d'instance de l'objet. Elles représentent l'état de l'objet.

PT1 est un pointeur sur la liste des noms de variables d'instance. Ces noms sont regroupés dans une structure de type ci-dessous :

var1	var2	....	varn

var1, var2, varn sont les noms des variables d'instance.

CEI est le compteur d'évolution de l'instance. Elle est utilisée pour vérifier la cohérence d'une instance par rapport à sa classe. Les descripteurs de toutes les instances à jour d'une classe possèdent le même champs PT1 :



2 SCHEMA D'APPLICATION D'UN MESSAGE

2.1 PROTOCOLE DE BASE

Lorsque l'analyse d'une expression APL détecte l'envoi d'un message "msg" à un objet OBJ, le mécanisme mis en oeuvre est le suivant :

i. On réquisitionne le descripteur de OBJ grâce à sa référence universelle qui se trouve dans la table des symboles de la WS-active ;

ii. on récupère l'identificateur de la classe, contenu dans le descripteur de OBJ soit OBJ-class ; on recherche ce nom dans la table des classes (liée à l'espace virtuel). Si le nom de la classe n'existe pas, on génère un message d'erreur. Sinon, cette table nous fournit la référence universelle de OBJ-class ;

iii. on réquisitionne le descripteur de la classe de OBJ, on compare les compteurs d'évaluation CEI de l'instance OBJ et CEC de la classe OBJ-class. En cas d'invalidité, on lance le protocole de mise à jour ;

iv. on recherche le sélecteur du message dans la table des méthodes de la classe. Si le sélecteur n'existe pas, on génère un message d'erreur ;

v. on sauvegarde le contexte de la WS courante. On active le contexte de la classe et on lance l'exécution ;

vi. à la fin de l'exécution, on libère les structures réquisitionnées et on restaure le contexte de la WS active.

L'application d'une méthode ne nécessite donc guère plus d'opérations qu'un simple appel de fonction. En outre il est possible d'optimiser certaines de ces manipulations (recherche de la classe, accès aux éléments de la classe ...) en utilisant le cache mémoire. Enfin notons que la généricité ainsi obtenue permet d'éviter des tests sur les opérandes, et donc de programmer des méthodes plus performantes.

2.2 LE PROTOCOLE DE MISE A JOUR

Il s'agit d'assurer à chaque instant d'utilisation, la cohérence d'une instance par rapport à sa classe.

Une classe peut subir des modifications même après sa création. Pour cela il suffit d'ouvrir la classe (chargement par la commande `)LOAD -CLIB`), d'effectuer les modifications et de refermer la classe (sauvegarde par la commande `SAVE -CLIB`).

Les modifications peuvent porter sur les variables de classe, les variables d'instance ou les méthodes. Ces modifications ont de l'influence sur deux catégories d'objets :

1. Les instances existantes de la classe
2. La descendance de la classe, autrement dit sur les sous-classes de la classe modifiée.

Le cas 2 est discuté dans la section relative à l'héritage.

Intéressons-nous donc aux instances existantes de la classe. D'après notre description interne de la classe, on constate que nous ne gardons pas de référence sur les instances d'une classe. Ce choix simplifie les structures de données et allège l'implantation. En effet en APL, la création et la destruction d'objets seront des opérations fréquentes.

Les objets sont traités comme les éléments primitifs d'APL. Nous ne garderons pas la liste des instances d'une classe. Notons que cette liste des instances ne serait d'aucune utilité pour la mise à jour. En effet dans le schéma d'application d'un message, la mise à jour est effectuée au moment ou l'on utilise une instance. Pour cela, on compare le compteur d'évolution, CEI, de l'instance avec celui de la classe de l'objet, CEC. En cas d'incohérence, l'instance est devenue obsolète. Cet état sera signalé par un message d'erreur. Dès lors l'utilisateur peut redéfinir l'instance ou le détruire. On peut ultérieurement concevoir un mécanisme de mise à jour automatique, déclenché par une telle erreur. Cependant, les avis dans le monde des L.O.O sont très partagés sur l'intèrêt d'un tel mécanisme.

En fait, seul le compteur CEC évolue avec les modifications de la classe. Mettre à jour une instance revient tout simplement à modifier le champ PT1 de son descripteur pour pointer sur la nouvelle liste des variables d'instance, CEI devient la copie de CEC.

Dans une optique où on ne compile pas les méthodes, il est facile de remarquer que seules les modifications touchant les variables d'instance font évoluer le compteur d'évolution CEC de la classe. En effet, on recherche à chaque fois le sélecteur de méthode dans la table des méthodes de la classe.

Outre PT1, CEC, la modification d'une classe peut entraîner une altération du contexte et de la liste des sélecteurs donc des champs PT2, PT3, PT4 et PT5.

L'intérêt du compteur d'évolution est de permettre la prise en compte globale de toutes les modifications intervenues sur la classe entre deux utilisations consécutives d'une instance, et donc d'éviter toute mise à jour inutile.

Notons que dans beaucoup de L.O.O. il n'est pas possible de modifier les champs d'une classe après sa création. Seules les méthodes peuvent être modifiées. La définition d'une classe est donc figée à sa création. Dans les L.O.O qui autorisent la modification des variables d'instances, il est très difficile de savoir les effets que cela entraîne. Ainsi dans Smalltalt-76, chaque instance est "cablée" génétiquement pour pouvoir ajouter ou retrancher un champs et sa valeur associée grâce aux méthodes :

+field| et -field|

La question qui se posait alors était de rajouter au non le couple <champ, valeurs> à toutes les instances de la classe ou uniquement à l'objet receveur. Là encore, un "effet de bord" vient perturber la définition du modèle, dans la mesure où il y aura dès lors des instances qui ne seront plus conformes à leur moule (classe).

3 A PROPOS DE L'HERITAGE

3.1 L'HERITAGE SIMPLE

D'après le postulat P 5, il existe une hiérarchie entre les classes. L'héritage simple implique que chaque classe soit sous-classe d'une seule et unique classe, qui sera appelée sa super-classe. Dans le descripteur de chaque classe, figure une identification de classe qui peut être le nom de sa super-classe.

RU	GDM	ID—SupClasse		CEC
PT1	PT2	PT3	PT4	PT5
val1	val2	. . . .		valn

L'intérêt de l'héritage est de réduire davantage la duplication de code code en permettant aux instances d'une classe d'accéder aux méthodes et aux champs de l'ascendance de cette classe.

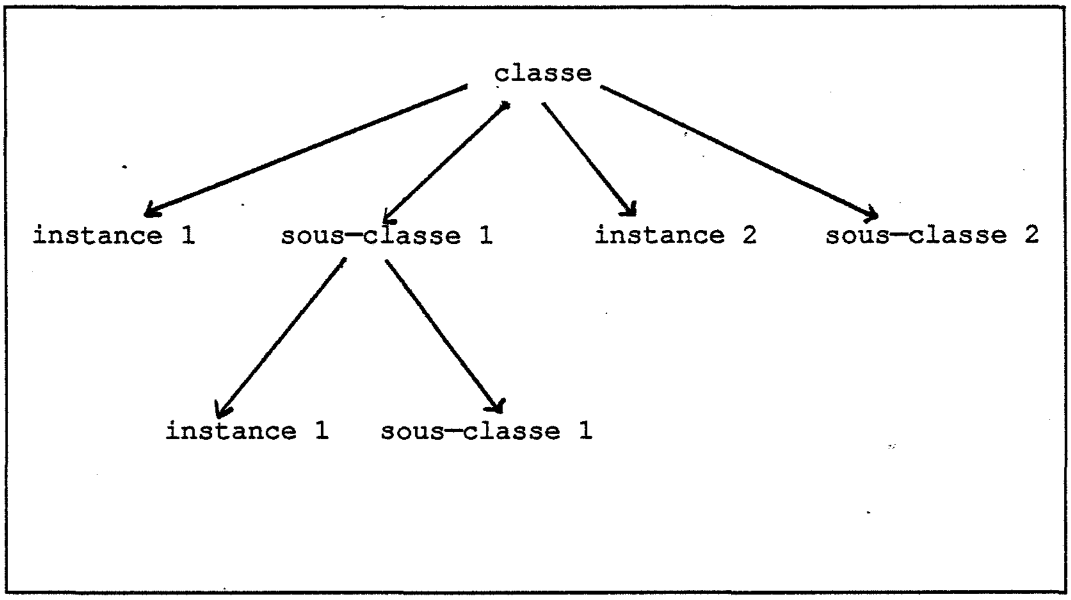
Il faut donc pouvoir déclarer une classe comme héritant d'une autre. Pour cela, nous proposons la fonction système ⎕*DC* en diadique. La syntaxe est :

'NOM_CLASSE' ⎕DC 'NOM_SUPER_CLASSE'

Ce qui crée l'argument gauche en tant que sous classe de l'argument droit.

La sous-classe sera initialisée à sa création avec une copie des noms et des valeurs des variables de classe et d'instance. Dès lors, elle devient une classe à part entière. On peut donc l'ouvrir, ajouter des champs ou des méthodes locales, créer des instances de cette nouvelle classe.

A une classe seront donc connectées génétiquement 2 entités : les instances et les sous classes. La vision d'une classe devient :



Le schéma d'application d'un message se modifie légèrement. En effet la recherche a lieu d'abord parmi les sélecteurs de la classe de l'objet, ensuite dans la super-classe de sa classe et ainsi de suite jusqu'à la première méthode qui concorde avec le nom du sélecteur ou bien jusqu'à épuisement de l'ascendance de l'objet.

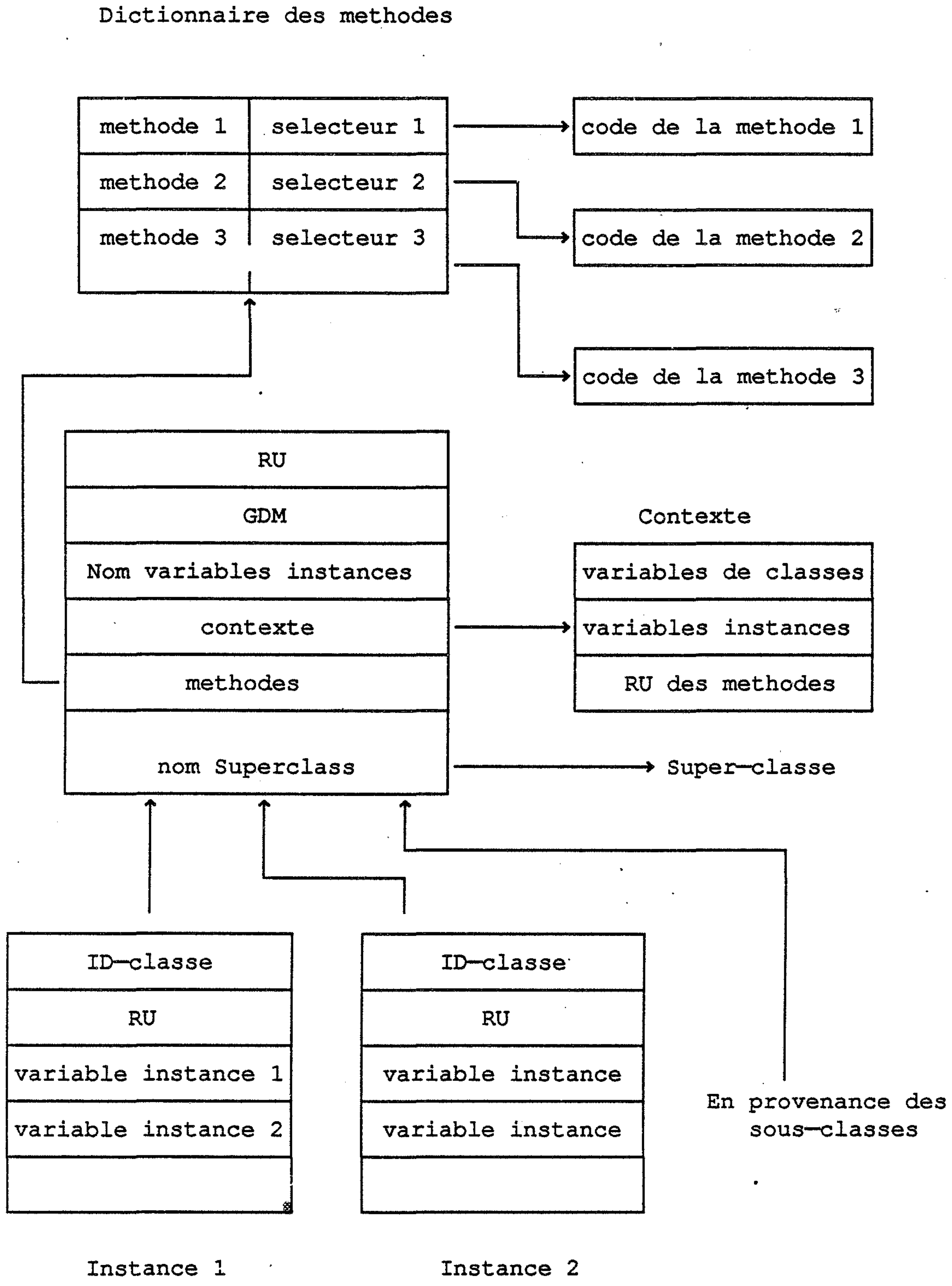
La classe de butée qui se trouve au sommet de la hiérarchie n'est pas sous-classe d'une autre classe. Son champs ID Superclasse contiendra NIL.

RU	GDM	NIL		CEC
PT1	PT2	PT3	PT4	PT5
val1	val2	. . . .		valn

NIL peut symboliser également la classe OBJECT dont on supposera conceptuellement qu'elle est la racine de la hiérarchie de toutes les classes.

Le protocole de mise à jour d'une instance ne varie pas, on se limite uniquement à la cohérence entre un objet et sa classe. Mais les modifications qui interviennent sur une classe altèrent-elles uniquement le compteur d'évolution de cette classe ou également les CEC de ses sous-classes ? Ce qui nécessiterait un mécanisme parallèle de mise à jour des sous-classes.

Une vision synthétique de la classe serait alors :



Si l'arbre d'héritage est relativement profond, la stratégie de balayage systématique des super classes à chaque transmission de message devient pénalisante.

Il faut alors se placer dans une optique de compilation des méthodes, ce qui revient à disposer au niveau de chaque classe des références sur toutes les méthodes auxquelles ses instances peuvent accéder. Si cela accélère l'application d'un message, en revanche, le protocole de mise à jour se complique. En effet, il faut signaler les modifications à toute la descendance de la classe.

3.2 HERITAGE MULTIPLE

Dans l'héritage multiple, le schéma d'application d'un message se complique singulièrement. Le parcours correspondant à l'algorithme de recherche s'effectue non plus dans une arborescence (parcours simple qui va va des feuilles à la racine) mais dans un graphe qui au demeurant, peut être cyclique.

Dans le contexte d'APL, une vision dynamique de l'héritage multiple nous semble plus appropriée.

En effet, nous avons déjà dit que les objets que nous créons seront relativement indépendants les uns des autres. L'héritage des champs de toutes les superclasses ne nous semble pas intéressant. Dans la mesure où une classe sera implantée comme une WS inactive, l'héritage multiple est un moyen adéquat pour accéder simultanément à des objets qui résident dans des WS différentes.

La vision dynamique de l'héritage consiste à admettre que chaque objet puisse redéfinir sa propre conception de l'héritage et non plus celle qui est attachée à sa classe. Dès lors on se rapproche d'un autre modèle objet dit des acteurs dans lequel un mécanisme de délégation remplace le mécanisme d'héritage.

Dans ce modèle, il n'est pas fait de distinction entre relation d'instanciation (entre une instance et sa classe) et relation de spécification (entre classes et superclasses). La distinction entre classe et instance n'a plus lieu d'être, le concept de classe étant absent du modèle. Toute entité est un générateur potentiel. Le mécanisme d'instanciation se fait par **copie différentielle** pour signifier qu'à la création d'un nouvel acteur, seules ses différences par rapport à l'acteur générateur seront exprimées, l'absence de différence se traduisant par une pure et simple copie.

Dans le système d'acteurs, le mécanisme d'héritage est remplacé par celui de la **délégation.**

Ce terme exprime que tout acteur ne sachant pas traiter un message (plutôt que de provoquer une erreur) transmet celui-ci à un autre acteur de ces accointances pour accéder de proche en proche à la méthode requise.

Au niveau de la réalisation, cette approche ne permet pas une factorisation des programmes aussi efficaces que celle des classes, en revanche, la problématique des métaclasses ne se pose plus.

Nous n'analyserons pas la dichotomie héritage/délégation en terme d'efficacité (comme c'est souvent le cas dans le monde des L.O.O), tant cette notion est vague et est fortement liée aux implémentations. Nous noterons simplement que dans l'héritage, il y a copie locale au niveau de chaque objet de toutes les variables d'instances de la hiérarchie de l'objet, même si ces variables ne seront pas toujours utilisées, mais qu'il n'y a pas de copie de méthode. Ainsi, dans ce système, l'espace occupé par un objet augmente beaucoup avec la profondeur de la hiérarchie. Dans ce système acteur, chaque objet a besoin de des propres variables et méthodes locales (copie différentielle) et de pointeurs sur un nombre (en général restreint) d'ascendants capable de répondre aux messages délégués. La taille occupée par un objet est indépendante de la profondeur de la hiérarchie.

Un de nos objectifs étant de structurer la WS APL et d'éviter de dupliquer le code des programmes, nous avons opté pour le modèle objet à la Smalltalt (vision avec les classes), mais le mécanisme de délégation nous semble être plus approprié à la vision dynamique de l'héritage. Il est intéressant de noter que le langage YAFOOL ("Yet Another Frame Based Object oriented Language") a réussi une synthèse entre les paradigmes de programmation orientée objet et de programmation orientée vers données. Au premier, il emprunte la notion d'objet. A chaque objet sont rattachés des attributs et comportements locaux dits **propriétés ou slots.** YAFOOL retient aussi (notamment des langages acteurs) la notion de **continuation** et de **bufférisation** de messages. La continuation est la structure de contrôle par laquelle un message indique par quoi il doit se "continuer". Du second, YAFOOL retient la notion de **réflexes** et les principes de base des **frames.** Les frames sont des entités semblables aux objets mais qui ne possèdent pas de procédures locales : ce sont leurs attributs qui les possèdent (d'où leur nom de **valeurs actives** dans les Flavors). Ces procédures locales sont appelées réflexes. L'héritage multiple de YAFOOL suit un mécanisme de délégation.

Il serait donc intéressant étudier des synthèses similaires dans le contexte d'APL.

CONCLUSION SECONDE PARTIE

A propos de notre modèle

Dans notre modèle toute entité n'est pas considérée comme un objet. En particulier :

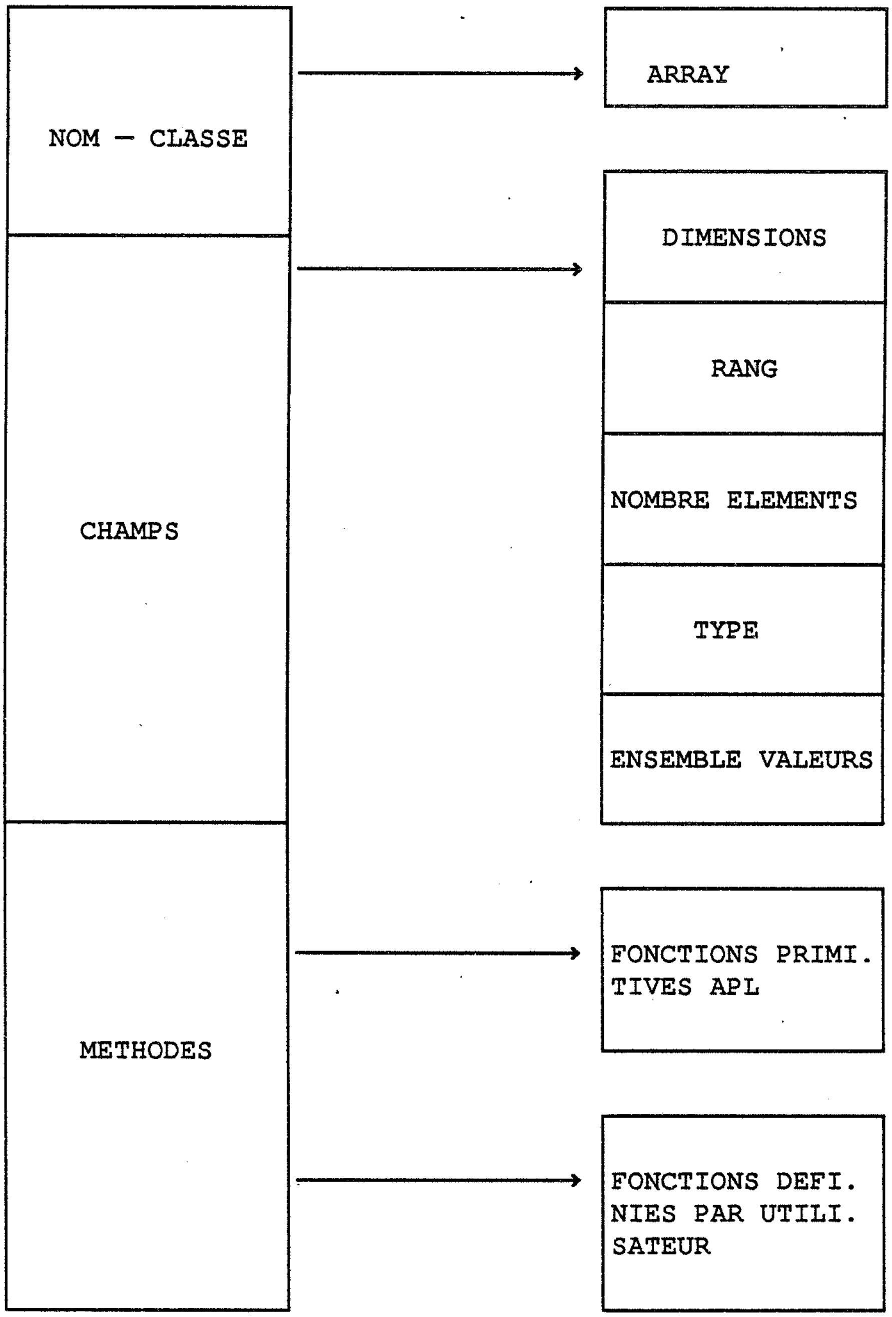
- les tableaux traditionnels du langage APL
- les classes ne sont pas des objets comme les autres. Elles ne sont pas instances d'une autre classe, et sont créés par la primitive ⎕*DC* ou par la sauvegarde d'une zone de travail.

Le principe d'uniformité nécessite que la seule structure de donnée soit l'objet. Dès lors peut-on voir un tableau APL ou une classe comme un objet ? Autrement dit peuvent-ils être générés à partir d'un objet modèle ?

En ce qui concerne les tableaux, de manière conceptuelle on est tenté de répondre par l'affirmative. En effet un certain nombre de parallèles peuvent être établis. Deux tableaux APL se distinguent uniquement par les valeurs d'un certain nombre d'attributs : le rang, le vecteur des dimensions, le nombre d'éléments, le type, l'ensemble des éléments constitutifs.

Ces tableaux lorsqu'ils se trouvent dans une même zone de travail partagent les mênes fonctionnalités que sont les actions définies par les fonctions primitives APL et les fonctions définies par l'utilisateur. On peut donc concevoir que les tableaux APL sont instances d'une classe que nous appellerons "ARRAY".

Une représentation de cette classe est :



Cette approche est relativement grossière. Tous les tableaux ne répondent pas à toutes les fonctions primitives par exemple la fonction + n'est pas définie pour les tableaux de type caractère. Il y a donc nécessité d'analyser de manière plus fine l'ensemble des tableaux afin d'exhiber une hiérarchie d'héritage. Ces classes seraient implicitement des classes prédéfinies dans OBJAPL 90.

conclusion seconde partie

Pour ce qui est des classes, l'incomplétude sémantique disparait si on respecte le postulat P4, une classe devenant elle-même instance de la classe **CLASS.**

Dès lors, un moule (classe) passe au rang de pièces ou exemplaires (instances) et est donc lui-même issu d'un moule, la classe CLASS qu'on appellera à ce titre **métaclasse.** On voit immédiatement surgir les dangers du raisonnement régressif. Le modèle Smaltalk-76 duquel on s'est inspiré arrête cette régression infinie en introduisant une circularité en haut de l'arbre d'instanciation, c'est dire que CLASS est instance (ou classe) d'elle-même. L'arbre d'instanciation devient :

L'inconvénient de cette vision est que l'existence d'une unique métaclasse (CLASS) impose un comportement commun à l'ensemble des classes.

Pour remédier à cela, Smaltalk 80 propose une généralisation de la métaclasse CLASS. Chaque classe possède sa propre métaclasse, qui spécifie les comportements propres à une classe donnée. La métaclasse sera utilisée essentiellement pour modifier ou construire de nouvelles méthodes d'instanciation ou d'initialisation. Ces méthodes opérant sur la classe elle-même, sont considérées à sa création, comme **méthodes de classes** (par opposition aux méthodes d'instances). Les arbres d'héritage et d'instanciation deviennent alors légèrement plus complexes.

Il est aisé de faire évoluer notre vision objet vers un modèle Smaltalk-80. Il suffirait pour cela d'uniformiser les fonctions du système ⎕*DC* et ⎕*NEW* d'une part, d'offrir un contexte d'exécution liée aux métaclasses d'autre part. Cette dernière condition se réalise en associant une table de symboles au descripteur d'une métaclasse.

Cependant, malgré les apparences, ce nouveau modèle ne respecte pas plus le principe d'uniformité. En effet, les métaclasses restent implicites et virtuelles :

- implicites, chaque métaclasse n'existe qu'à travers la spécification des méthodes de classe et n'est pas créée explicitement en tant que classe comme les autres par l'utilisateur ;

- virtuelles, une métaclasse n'est pas réellement accessible à l'utilisateur (par exemple, on ne peut pas l'instancier, son unique instance étant la classe qui l'aura induite).

Ces métaclasses bénéficient par ailleurs d'une hiérarchie parallèle à leurs classes, mais elles n'ont pas un statut de classes comme les autres et ne sont pas directement accessibles à l'utilisateur. Une classe n'a donc toujours pas un statut d'instance à part entière, son générateur (métaclasse) étant créé à partir d'elle et non l'inverse.

Dans [Briot85], on trouvera la définition d'un nouveau modèle qui donne à la classe un vrai statut d'instance. Ce nouveau modèle est en fait une simplification du modèle Smaltalk. L'auteur introduit un générateur de classes appelé METACLASS et un générateur d'instances terminales appelé CLASS. Outre sa simplicité, ce modèle évite le recours à des notions supplémentaires telles les méthodes de classe et les variables de classe, réduisant du même coup le nombre de concepts, tout en augmentant leur clarté. L'implantation intégrale de ce modèle se trouve dans [Briot84].

Le futur de la P.O.O en APL

Il est difficile de tirer une conclusion d'une expérience isolée. Mais la plupart des concepteurs de L.O.O sont d'accord pour dire qu'un système orienté objet n'est pratique que s'il offre beaucoup de classes prédéfinies, manipulables directement par l'utilisateur ou qui peuvent lui servir comme modèles pour faire ses propres extensions. C'est le premier objectif auquel il faut arriver rapidement.

D'autre part la comparaison de notre système avec Smalltalk 80 montre que la plupart des classes prédéfinies de Smalltalk (telles que les diverses classes de nombres, caractères, chaînes, tableaux, et même des ensembles) sont nécessaires pour offrir des fonctionnalités que l'on rencontre dans le système APL classique. Par exemple il existe dans Smalltalk une classe "ARRAY" mais ces concepts et sa réalisation sont si pauvres comparativement aux tableaux dans APL, qu'il n'y a aucun intérêt à imiter cette technique dans le cadre d'APL.

L'intégration des méthodes sous forme de fonctions machines (au moins pour les classes prédéfinies) sera une source d'efficacité pour la manipulation des objets.

Vue la puissance offerte par les tableaux APL, il est probable que dans APL un objet sera plus un sous-système spécialisé quasi indépendant, qu'une entité simple (entier, caractère, ...). Par exemple, [Soop84] propose d'utiliser les zones de travail comme base de données. Dans cette optique, notre système offre la notion de classe qui permet la gestion de la base de données par des algorithmes spécialisés (méthodes), et les intances seront des partitions de la base de données dans des modules indépendants. Cette approche permet l'isolation de la base de donnée vis à vis des accès abusifs.

L'extension objet pourrait être efficace dans le domaine du graphique où les objets sont complexes mais assez bien structurés, ou pour unifier les dialogues avec les processeurs auxilliaires des systèmes APL.

A l'heure actuelle, le modèle OBJAPL 90 est une synthèse entre les paradigmes de programmation orientée procédures (car on conserve les mécanismes du langage APL natif) et de la programmation orientée objet (dans la mesure où on introduit les concepts d'objets, de classe etc.).

CONCLUSION

Nous avons cherché à présenter dans les pages précédentes quelques uns des aspects les plus originaux du système APL 90.

L'étude des systèmes APL existants nous a permis de définir un modèle d'architecture logicielle de machine pour les langages de traitement de. tableaux (ALM 90). Sur cette architecture, nous avons réalisé un interprète complet en accord avec la norme ISO du langage APL. Ceci a constitué la version 1 du système APL 90. La recette de cette version a eu lieu à l'Agence De l'Informatique au cours du mois de mars 1985 et a fait l'objet d'un rapport d'expertise.

Nous avons montré que cette architecture bâtie autour d'un univers unique d'objets, intégrant la notion de référence universelle, permet de pallier certaines faiblesses du langage, et offre une grande souplesse pour l'extension du système et son ouverture vers l'extérieur. Le souci d'optimisation de l'accès aux objets, ainsi que le désir de faciliter leur manipulation, nous ont conduit à définir une hiérarchie de gestion mémoire[Sako83], dont on trouvera une description en annexe 1, distinguant la gestion de la mémoire virtuelle brute de celle des blocs partageables. Dans ce contexte, la technique des blocs arborescents et l'usage des compteurs de références se sont révélés efficaces pour la manipulation des structures de données utiles à tout interprète APL. De surcroît, une telle gestion s'est avérée bien adaptée au développement de l'extension objet.

Notre but étant de réaliser un système propre à l'expérimentation, nous ne pouvions ignorer les propositions d'extensions (que nous avons qualifiées de classiques), émanant de la communauté APL. Leur analyse selon des critères qui nous paraissent importants (compatibilité, formalité, simplicité, universalité, etc.), nous a permis de retenir les propositions les plus significatives. Cette analyse a nécessité une recherche bibliographique importante, qui nous a amené à classer ces extensions selon les points de vue de deux "écoles", et à analyser leurs fondements ainsi que leurs motivations. Force est de constater que les deux conceptions sont inconciliables, tant elles émanent d'approches radicalement différentes. Partant de cela, nous avons décidé dans un premier temps de faire évoluer notre système en intégrant des extensions compatibles avec celles d'APL 2 d'IBM.

Les travaux de Girardot et de Rollin [Girardot86, Rollin83] nous ont beaucoup éclairé sur les méthodes d'analyse des expressions APL, et en particulier sur les mécanismes syntaxiques et sémantiques des extensions.

La majeure partie de ces extensions a donc été intégrée au système APL 90, intégration facilitée par les choix faits pour la gestion de la mémoire. Par exemple l'introduction des tableaux généralisés et leur prise en compte par l'ensemble des fonctions primitives de restructuration n'a nécessité que quelques heures de mise au point.

La recherche d'un environnement de programmation de plus en plus convivial, nous a conduit à nous intéresser à la programmation orientée objet. L'extension orientée objet du langage APL a nécessité la description des concepts sur lesquels se fondent les L.O.O (objet, classe, instance, transmission de message...). Nous avons montré d'une part que ces concepts recouvrent des formes très variées dans la pratique, d'autre part qu'il existe différents modèles objets. L'examen attentif de quelques langages étendus aux objets révèle que chaque langage a une vision spécifique de la P.O.O. C'est la raison pour laquelle il n'est pas aisé d'énoncer des critères exhaustifs de classification des L.O.O.

Après avoir rappelé quelques tentatives montrant la nécessité d'introduire l'abstraction de type de donnée dans APL et précisé nos motivations, nous avons choisi pour notre extension orientée objet le modèle Smalltalk qui s'énonce facilement en cinq postulats. L'étude détaillée de ces postulats, montre qu'il ne peuvent pas être transférés tels quels d'un contexte Smalltalk à celui d'APL. Les tentatives d'intégration au système APL 90 de certains de ces postulats, nous ont permis d'élaborer le modèle OBJAPL 90. De cette expérience, nous tirons les constatations suivantes :

- en APL il y a une différence fondamentale entre structure de donnée et type de donnée, et un objet défini doit être considéré comme un type de donnée, quelles que puissent être les structures de données utiles à sa matérialisation (instanciation)

- le statut le plus approprié pour un objet défini est d'être atomique.

Cependant, pour respecter la philosophie d'APL, le traitement correct de l'encapsulation des procédures et des données nécessite l'usage de noms distingués pour l'identification des méthodes à invoquer. Ceci nous a conduit aux choix suivants :

- une classe s'implante comme une zone de travail particulière, et peut être manipulée comme telle,

- la transmission d'un message à un objet revient en définitive à l'activation d'une fonction définie.

Dans la pratique :

- La manipulation des objets dans APL soulève beaucoup de questions. Les réponses à ces questions ne nécessitent dans certains cas que des modifications mineures dans le système APL 90 : ajout de règles pour l'analyse syntaxique, ajout de quelques nouvelles variables et fonctions du système, ... Mais dans d'autres cas ces modifications sont très importantes : application des fonctions primitives sur les objets et les procédures de conversion par exemple.

- Les objets doivent satisfaire des contraintes de propriétés liées à des fonctions primitives comme ρ et ≡.

- les structures de données nécessaires à la gestion des objets s'appuient fortement sur celles déjà existantes dans le système non étendu.

Au terme de ce travail nous soulignerons les points suivants :

- la P.O.O permet en APL un meilleur partage de code, la définition de nouveaux types de donnée par l'utilisateur et apporte une certaine structuration dans l'espace de travail APL.

- La synthèse entre la programmation orientée objet et la programmation procédurale peut engendrer dans APL un nouveau style de programmation puissant et très modulaire.

- Les concepts des L.O.O s'intègrent dans un environnement APL, moyennant certains choix, sans remise en cause de l'esprit et des mécanismes fondamentaux du langage et du système.

Le choix du modèle Smalltalk résulte de la simplicité de ses concepts. Dans le cadre du projet APL 90, nous cherchons dans un premier temps à proposer un outil expérimental. De même que dans Smalltalk, cet outil devra avoir au fur et à mesure de son évolution, un certain nombre de classes systèmes (ou classes prédéfinies), à partir desquelles l'utilisateur pourra créer ses propres objets.

Il s'agit donc d'une extension conservatrice, destinée à induire des réflexes nouveaux chez le programmeur APL. OBJAPL 90 n'est pas un choix "philosophique" définitif de modèle-objet. C'est pourquoi il peut être utile d'analyser de plus près le modèle acteur dans lequel la distinction entre classes et instances devient sans objet, et dont le mécanisme de délégation pourrait offrir des possibilités intéressantes. En tout état de cause, comme aucun autre système APL existant (à notre connaissance) n'offre de possibilité de programmation orientée objet, il paraît important, de passer à une phase pratique, et d'offrir cet outil expérimental à d'autres utilisateurs, afin de parvenir à une évolution du modèle que nous avons défini.

En conclusion, nous pouvons dire que les objectifs immédiats que nous nous étions fixés dans le cadre du projet APL 90 à savoir :

- définition d'un interprète APL respectant la norme ISO,
- extension du système vers une compatibilité avec APL 2 d'IBM,
- introduction des concepts des L.O.O dans APL,

ont été atteints.

BIBLIOGRAPHIE

[Abelson85] : **Abelson H., Sussman G.J.,** *Structure & Interpretation of Computer Programs.* The MIT Press, Cambridge (Massachussets), London (England).

[Abrams70] : **Abrams P.S.** *An APL machine* Ph.D Thesis, Standford University, feb 1970.

[Alfonseca76] : **Alfonseca M., M.L. Tavera** *Extension of APL to tree-structured Information.* APL76, 1-23. (SCF)

[Amade-Girardot-Mireaux-Sako86] **Amade B. P., Girardot J. J., Mireaux F., Sako S.** *APL 90 : Le manuel de référence.* Ecole des Mines et Société SYNC, Saint-Etiènne

[Baron82] S. **Baron** *"Les tableaux généralisés"* actes congrès AFCET, La Programmation en APL Avril 1982.

[Benoit85] : **Benoit Ch., Pherivong Ch.,** *Le modèle objet : analyse et évolution,* pp 45-52, Bigre + Globule n. 45, Octobre 85.

[Bernecky80] : **Bernecky B, Iverson K.E.,** *Operators and enclosed arrays.* In C80, pp. 319-331.

[Berry79] : **Berry Paul** *SHARP APL reference manual.* I.P.SHARP Associates, Inc March 1979.

[Bertin81] : **Bertin Christian** *Aspects de la réalisation d'un système APL optimisé.* Thèse de Docteur Ingénieur Décembre 81- Ecole des Mines de Saint-Etienne.

[Birtwistle73] : **Birtwistle G., Dahl O., Myhrhaug B., Nygaard K.,** *SIMULA BEGIN,* Petrocelli/Charter, New York (1973)>

[Borning82] : **Borning A., Ingalls D.M.,** *Multiple inheritence in Smalltalk-80,* Proc of AAAI-82, pp 234-237, Pittsburgh (1982).

[Brachman83] : **Brachman R.,** *What Is-A is and isn't : an analys of taxonomie links in semantic networks,* IEEE Computer, pp 30-36, October 1983.

[Braffort] : **Braffort P., Michel J.,** *X\APL : An experimental extensible programming system.* Tech. Report Mathematique Université, Paris XI, Orsay, France.

[Brassard86] : **Brassard G., Monet S., Zuffellato D.,** *Algorithmes pour l'arithmétique des très grands entiers,* TSI, Vol. 5, n.2, 1986.

[Briot84] : **Briot J.P.,** *Instanciation et héritage dans les langages objets,* LITP 85-21, Paris 6ème, Thèse de 3ème cycle, Décembre 1984.

[Briot85] : **Briot J.P.,** *Les métaclasses dans les langages objets,* 5ème congrès AFCET-RFIA, Grenoble, Novembre 1985.

[Brown71] : **Brown J.A** *A Generalization of APL* Doctoral Thesis, 1971, Syracuse University, New York.

[Brown79] : **Brown J.A** *Evaluating APL Extensions* APL79, pp 148-155 ACM

[Brown81] : **Brown J.A, M. A. Jenkins,** *The APL identity crisis* In APL79, ACM

[Chailloux85] : **Chailloux J.,** *Le-Lisp,* Version 15, Manuel de référence, INRIA, Février 1985.

[Chomat71] : **Chomat P.** *Notions sur les systèmes APL/360.* Grenoble, IUT Informatique, sept 1971.

[Cointe83a] : **Cointe P.,** *A VLISP Implementation of SMALLTALK-76,* pp. 89-102. Integrated Interactive Computing Systems, ed. P. Degano & E. Sandewall, North-Holland, Amsterdam, New-York, Oxford (1983).

[Cointe84a] : **Cointe P.,** *Une extension de VLISP par les objets,* Science of Computer Programming 4, (161) pp. 291-322, North-Holland (1984).

[Cointe84b] : **Cointe P.,** *Implémentation et interprétation des langages objets, application aux langages Formes, ObjVLisp et Smalltalk* (Thèse d'Etat) LITP 85, Univeresité Paris VI, IRCAM, Décembre 84.

[Cointe85] : **Cointe P.,** *Le modèle OBJVLISP une plate forme pour l'expériementation des formalismes objets,* 5ème Congrès AFCET-RFIA Grenoble (25-27 Novembre 1985).

[Cointe86] : **Cointe P.,** *Une introduction à la programmation par objet.* Giens 86 : deuxièmes journées Bases de Données Avancées.

[Dahl70] : **Dahl O., Myhrhaug B., Nygaard K.,** *SIMULA-67 common base language,* S-22, Norvegian Computing Center, Oslo (October 1970), Simula Information.

[Dave82] : **Dave Rabenhorst & Alii** *APL2 reference manuel* IBM Corporation.

[Ducournau86] : **Ducournau R., Quinqueton J.,** *YAFOOL : encore un langage objet à base de frames !* Rapport technique numéro 72 INRIA, Aout 1976.

[Edwards73] : **Edwards E.M.** *Generalized arrays (lists) in APL.* In C73, pp. 333-338.

[Fabry74] : **Fabry R.S** *Capability based addressing* in C ACM, July 74, 174,pp 403-412.

[Falfoff81] : **Falkoff Adin** *More on strand notation* APL Quote Quad vol 2 num 2 décembre 1981.

[Falkoff68] : **Falkoff Adin D and Iverson Kenneth E.** *APL/360 user's manuel.* IBM Corporation 1968.

[Falkoff73a] : **Falkoff A.D., Iverson K.E** *The design of APL.* IBM J.Res. Develop., vol.17, num 4, July 1973.

[Falkoff73b] : **Falkoff A.D., Iverson K.E** *APL : manuel de référence,* traduit par Pinta et Leborgne, IBM, GHF2-0056, 1973

[Falkoff73c] : **Falkoff A.D., Iverson K.E** *APL-SV user's manual.* IBM, SH-1460, 1973.

[Falkoff79] : **Falkoff A.D., Orth D.L.** *Development of an APL standard.* In C79, pp.409-453.

[Falkoff81] **A. Falkoff** *"More on strand Notation"* in APL Quote-Quad, vol 12-2, December 1981.

[Finger81] : **V. Finger & G. Médigue** *Architecture multiprocesseur et disponibilité : la SM 90.* L'echo des recherches num 105, juillet 1981.

[Forkes81] : **Forkes, Doug.** *Complex Floor Revised.* APL81, 107-111

[Gal 76] : *Primitive fonctions for graphics in APL.* In APL quote Quad vol 7 num 1 summmer 1976 pp 27-36

[Gautron86] : **Gautron Ph.,** *C++ : Instanciation, héritage et transmission de messages.* Bigre + globule n. 48, Janvier 1986.

[Ghandour73] : **Ghandour Z., Mezei J.** *General arrays, operators and functions.* IBM journal of res. & develop., July 1973.

[Girardot-Mireaux76] : **Girardot JJ, Mireaux F.** *Réalisation d'un interprète complet du langage APL sur un miniordinateur.* Thèse de Docteur-Ingénieur, Université de Nancy I, sept 1976.

[Girardot82] : **Jean-Jacques Girardot** *Architecture logicielle et matérielle d'une machine APL.* Rapport de recherche - EMSE - sept 1982.

[Girardot85] : **Jean-Jacques Girardot** *The APL 90 Project : new directions in apl interpreters technology* In APL85 Conference Proceedings ACM, Seattle may 85.

[Girardot86] : **Girardot J.J., Rollin F.,** *The syntax of APL : an old approach revisited.* Soumis à publication dans ACM 87, Conference Proceedings, APL in Transition, Dallas, Mai 87.

[Girardot-Mireaux-Sako85a] : **Girardot J.J., Mireaux F., Sako S.,** *APL 90 : Manuel de maintenance* Département Informatique de l'Ecole des Mines de Saint-Etienne.

[Girardot-Mireaux-Sako85b] : **Girardot J.J., Mireaux F., Sako S.,** *APL 90 : rapports techniques.* Département Informatique de l'Ecole des Mines de Saint-Etienne.

[Girardot-Sakellaridis-Sako83a] : **Girardot J. J., Sakellaridis U., Sako S.,** *Pourquoi un cours APL et comment initier à l'informatique ?* Actes des Journées Afcet-APL, Avril 1983.

[Girardot-Sakellaridis-Sako83b] : **Girardot J. J., Sakellaridis U., Sako S.,** *L'impact du cours APL et les enseignements à en tirer* Actes des Journées Afcet-APL, Avril 1983.

[Girardot-Sako85a] : **Girardot J.J., Sako S.,** *APL sur SM 90 : le système APL 90* Actes des journées SM 90, Verasilles, Décembre 1985 (Adi)

[Girardot-Sako85b] : **Girardot J.J., Sako S.,** *APL 90* Dans USR/SM 90, num 9, Spécial Langages.

[Girardot-Sako86] : **Girardot J.J., Sako S.,** *An object oriented extension to APL* Soumis à publication dans ACM 87, Conference Proceeding, APL in Transition, Dallas, Mai 87 USA.

[Goldberg83] : **Goldberg A., Robson D.,** *SMALLTALK-80, the language and its implementation,* Addison Wesley Publishing Company.

[Goldberg84] : **Goldberg A.,** *SMALLTALK-80, the interactive programming environnement.* Addison Wesley Publishing Company.

[Greussay78] : **Greussay P.,** *Le système VLISP-16,* Ecole Polytechnique, Décembre 1978.

[Guiboud75] : **Serge Guiboud-Ribaud** *Mécanismes d'adressage et de protection dans les Systèmes. Application au noyau GEMAU.* Thèse Etat. Univ, Grenoble, juin 75.

[Gull79] : **Gull W.E., Jenkins M.A.,** *Recursive data structures in APL.* In publ. of the ACM, Vol. 22, n.1, Jan. 1979,pp. 79-96.

[Habib86] : **Habib M., Bouchitte V., Hamroun M., Jégou R.,** *Depth-first search and linear extensions.* Laboratoire d'informatique de Brest, Février 1986.

[Hardwick81] : **Hardwick M.,** *Graphical Data Structures in APL.* APL81, 129-136, (PrF, NA).

[Hassit73] : **Hassit et Lyon** *Implementation of high level language machine.* Communications of the ACM - April, vol 16 numb. 4

[Hassit76] : **A Hassit & L.E. Lyon** *An APL emulator on System 360*

[Hewitt75] : **Hewitt C.E, Smith B.,** *A PLASMA Primer.* MIT Artificial Intelligence Laboratory, September 1975.

[Hewlett77] : *Hewlett packard Journal* july 1977.

[IBM 73] : *TSIO program reference manuel.* IBM, ref. SH 20-1463, 1973.

[Iverson-Brown-Smith81] **K. Iverson, B. Smith, J. Brown** *"Correspondance on Strand Notation"* in APL Quote-Quad, vol. 11-3, March 1981.

[Iverson62] : **Iveron K.E.** *A programming language.* New-York, Wiley, 1962

[Iverson71] : **Iversion K.E** *Abrega as a Language.* In C71, pp.5-16.

[Iverson73] : **Iverson E. B.** *APL/4004 implémentation.* In c73, pp231-236.

[Iverson78] **K. Iverson** *"Opérators and Functions"* Research Report 7091, IBM Corporation, 1978.

[Iverson80] **K. Iverson, B. Bernecky** *"Operators and Enclosed Arrays"* in proceeding of the IPSA 1980 APL Users Meeting.

[Jenkins80] : **Jenkins J.M.,** *ALICE : An extensible language based on APL concepts.* QUTR 80-104, Nov. 1980.

[Jenkins80] : **Jenkins, Jean Michel** *ALICE: An Extensible Language Based on APL Concepts.* QUTR Novembre 80

[Juran74] : **Juran, Moore, Orndorff, Rice** *pcs shared file system.* in APL Quote Quad, vol 5 num 1/3 Spring 1974, pp 5-16.

[Kajiya81] : **Kajiya J.,** *Generic functions by non-standard name scoping in APL.* APL81, 172-179,.

[Kay81] : **Kay Alan C.** *Generic Programming : APL and Smalltalk* In ACM proceedingsAPL81, 172-179,.

[Langlet82a] : **Gérard Langlet.** *Pourquoi utiliser préférenciellement les vecteurs dans une programmation APL.* In La Programmation en APL. AFCET, 22-23 avril 1982.

[Legrand79] : **Legrand B.** *Apprendre et appliquer le langage APL.* Masson,1979.

[Lenfant82] : **Jacques Lenfant.** *L'informatique japonnaise sur les traces de l'industrie automobile.* in TSI 1 num 3, mai-juin 1982.

[Liskov84] : **Liskov B.,** *Overview of the argus language and system,* 1984.

[Macon74] : **Macon H.P** *Virtual APL* in APL Quote Quad vol 5 num 1/3 , springs 1974 pp 17-23

[Martin72] : **Martin H, Levy E. Raynaud.** *1ère partie : Le langage. 2ème partie : L'interprétation.* Thèse de Docteur-Ingénieur, Toulouse, Université P. Sabatier mai 1972.

[Mcdonnel73] : **Eugene. E. Mcdonnell** *Complex Floor* APL Congress 1973

[Minsky75] : **Minsky M.,** *A framework for representing knowledge.* In P.H. Winston, editor, The psycology of Computer Vision, pages 211-277, Mc Graw-Hill Computer Series, New-York, St-Louis, San-Francisco, 1975.

[Moon 81] : **Moon D., Weinreb D.,** *Flavors Message Passing in the Lisp Machine,* MIT AI Lab. A.I., memo n. 602 (1980) et Lisp machine manual.

[More73] : **More T.,** *Axioms and theorems for a theory of arrays.* IBMJRD v17 2, 135-175, March 1973, (AT, NA).

[More79] : **Moret T.,** *The nested rectangular array as a model of data.* In C79, pp. 55-73.

[Murray73] : **Murray, Ronald C** *On Tree Strucrure Extensions to the APL Language.* ACM APL73

[Nakache76] : **Nakache M.,** *Réalisation d'un noyau de système de gestion de base de données relationnelles sous APL.* Thèse de Docteur-Ingénieur, Saint-Etienne, Ecole des Mines, Juin 1976.

[Norme86] : **ISO/TC 97** *Programming languages - APL* Draft international standard ISO/DIS 8485.

[Orgass77] : **Orgass R.,** *The 1E6?1E6 APL Workshop : Another overview.* APLQQ v82, 8-11, Dec. 1977, (NA, Nam).

[Ouanes78] : **Ouanes M.** *APLIXI, opérateurs graphiques, manuel de référence* APLIXI, Paris 1978.

[Pakin75] : **Pakin S., Polivka R.P.** *APL : the language and its usage*

[Pierre79] : **Pierre M., Pierre P.,** *A relationnal data base using generalized arrays and data base primitives.* In C79, pp. 102-109.

[Rentsch81] : **Tim Rentsch** *Object-oriented programming,* Sigplan notices vol 17, 9, 81.

[Robichaud77] : **L.P.A Robichaud** **\APL : An Extensible APL System* Université de Laval Québec Aout 1977.

[Rollin83] : **Florence Rollin** *Syntaxe et Analyse des Langages Dérivés d'APL,* Mémoire de D.E.A septembre 83 I.N.P.G et Ecole des Mines de Saint-Etienne.

[STSC73] : *APLPLUS Workspace documentation. User's guide.* Bethesda, Maryland, Scientific Time Sharing Corporation, 1973.

[Sako83] : **Sako Sega** *Aspects de la gestion mémoire du système APL 90* Mémoire de D.E.A septembre 83 I.N.P.G et Ecole des Mines de Saint-Etienne.

[Sako84] : **Sako Sega** *APL 90 et la SM 90* Actes congrès AFCET-APL mai 84, Liège mai 1984 Belgique.

[Sako86] : **Sako Sega** *Une extension orientée objet d'APL* Actes congrès AFCET-APL juin 86, Saint-étienne.

[Serpette84] : **Serpette B.P.,** *Contextes, processus, objets, Séquenceurs : FORMES,* LITP 85, Université Paris VI, Paris (Octobre 84), Thèse de 3ème cycle.

[Smith81] : **Smith R.A.,** *Nested arrays.* APL Plus, S.T.S.C., 1981.

[Snyder79] : **Snyder Alan** *A Machine Architecture to Support an Object-Oriented Language.* PHD March 1979 M.I.T

[Soop84] : **Karl Soop,** *Can an APL workspace be used as a data base.* In APL quote-quad 14, 4, June 1984.

[Vasseur73] : **Vasseur J.P.,** *Extension of APL operators to tree-like data structures.* In C73, pp. 457-464.

[Wertz83] : **Wertz H.,** *Filtres, contextes et démons.* Education & Informatique, (16) : 39-42, Mai-Juin 1983>

[Wiedmann79] : **Clark Wiedmann** *Steps toward an APL compiler.* in APL Quote Quad vol. 9 num 4 juin 1979.

[Winston81] : **Winston P.H., Horn B.K.,** *LISP,* Addison Wesley, 1981.

ANNEXE 1

1 INTRODUCTION

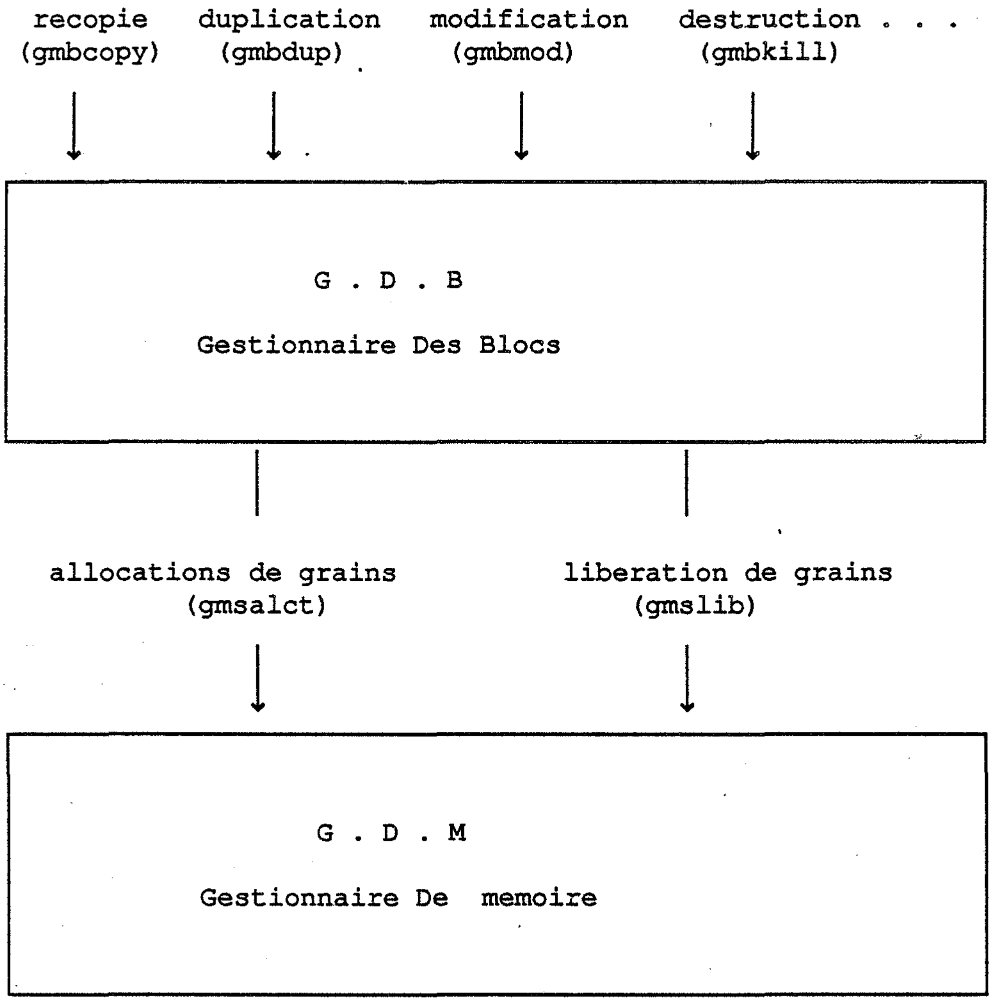
Cette annexe décrit quelques aspects de la gestion de la mémoire virtuelle d'APL 90. Les allocations et désallocations de la mémoire virtuelle se font à deux niveaux.

- Le premier niveau est la gestion classique les blocs de taille variable de la mémoire

- Le second niveau se consacre aux différents partages pouvant se faire sur ces blocs, plus particulières à la gestion des cohérences de ces partages pour les blocs contenants des pointeurs sur d'autres blocs. La mémoire occupée apparaît alors comme un ensemble de blocs structurés sous la forme d'un graphe orientée sans circuit.

Tous les objets manipulés par l'interprète sont définies par une référence universelle. Le partage de la même information est rendue possible en admettant plusieurs pointeurs sur un même blocs. Il est alors nécessaire d'associer à chaque blocs un **compteur de référence,** indiquant le nombre de "propriétaires' qui possèdent cet objets. Un tel bloc ne peut être effectivement détruit (rendu à l'allocataire du premier niveau) que lorsque le compteur associé devient nul.

On obtient alors pour la gestion de la mémoire la hiérarchie suivante :



2 GESTION DE LA MEMOIRE SECONDAIRE

2.1 STRUCTURE D' UNE REFERENCE UNIVERSELLE

Il existe 2 types de **référence universelle :** celles qui sont autodécrites et celles qui contiennent une adresse universelle. Dans les 2 cas la taille de la référence universelle doit être la même. Nous avons fixé cette taille à 64 bits. Ce qui permet d'une part d'autodécrire plusieurs types d'objets, d'autre part d'avoir un grand espace adressable.

La distinction entre les 2 types de référence universelle est effectuée par le bit 0 du premier octet :

0xxx xxxx	référence auto-décrite
1xxx xxxx	référence avec adresse

Une référence universelle est définie par la structure suivante :

```
typedef struct {
        char    typ1,typ2;
        short   idev;
        long    adr;
} RU ;
```

Le premier octet (typ1) définit principalement le rang de l'objet, on distingue:

- scalaire	RUTSCAL
- vecteur vide	RUTVID
- vecteur court	RUTVCRT
- vecteur normal	RUTVNORM
- tableau	RUTTAB
- commentaire	RUTCOM
- identificateur	RUTID
- unité syntaxique	RUTUS
- objet défini	RUTOBJ

Le deuxième octet (typ2) donne son type et dans le cas des vecteurs courts la longueur :

00 à 7F	logique
80 à 8F	caractère 8 bits
90 à 97	caractère 16 bits
98 à 9D	caractère 32 bits
A0 à AF	entier 8 bits
B0 à B7	entier 16 bits
B8 à BD	entier 32 bits
C0 à C3	réel 32 bits
C8 à C9	réel 64 bits
D0	complexe
E0 à E1	référence universelle

2.1.1 REFERENCE UNIVERSELLE AUTODECRITE

Elle contient l'objet, et lui sert de mini-descripteur. Plusieurs types d'objets peuvent être concernés.

Les 48 derniers bits de la référence contiennent la valeur. Dans le cas des vecteurs courts, le second octet d'une référence universelle contient la longueur du vecteur. Ainsi on a :

type logique:

0xxx xxxx 0-127 ==> 1 à 128 éléments

type caratère 8 bits:

1000 xxxx 0-15 ==> 1 à 16 éléments

type caractère 16 bits:

1001 0xxx 0-7 ==> 1 à 8 éléments

type caractère 32 bits:

1001 10xx 0-3 ==> 1 à 4 éléments

type entier 8 bits:

1010 xxxx 0-15 ==> 1 à 16 éléments

type entier 16 bits:
1011 0xxx 0-7 ==> 1 à 8 éléments

type entier 32 bits:

1011 10xx 0-3 ==> 1 à 3 éléments

type réel 32 bits:

1100 00xx 0-3 ==> 1 à 3 éléments

type réel 64 bits:

1100 010x 0-1 ==> 1 à 2 éléments

type complexe:

1100 1000 1 élément

type référence universelle:

1110 000x 0-1 ==> 1 à 2 éléments

Si la valeur du tableau tient sur 48 bits elle est placée dans la référence universelle. Ainsi un vecteur auto-décrit peut donc contenir:
48 logiques
6 caractères ou entiers 8 bits
3 caractères ou entier 16 bits
1 caractère, entier, ou réel 32 bits

2.1.2 REFERENCE UNIVERSELLE NON AUTODECRITE

Les 48 derniers bits contiendront une adresse universelle mémoire secondaire qui se décompose comme suit :

- le champ **idev** est l'identification d'espace virtuelle. Sa longueur est de 16 bits. Elle permet d'accéder à des objets d'un espace différent de l'espace courant.

- le champ **adr** est une adresse universelle de de bloc mémoire qu'on appellera **grain.** Sa longueur est de 32 bits.

Le fait qu'un tableau soit optimisé est indiqué par le bit 1 du premier octet de la RU :

10xx xxxx tableau non-optimisé
11xx xxxx tableau optimisé

Seuls les tableaux avec référence mémoire sont optimisables (c'est-à-dire les vecteurs normaux ou les tableaux de rang supérieur à 1).

2.2 STRUCTURE D'UN BLOC MEMOIRE

Un bloc mémoire débute par un en-tête de 16 octets qui servent aux primitives de gestion mémoire. A l'heure actuelle 2 types de bloc sont gérés par APL 90.

Les blocs sans pointeur

Ils ont l'entete suivant :

copie de la RU	: 8 octets
semaphore d''acces	: 1 "
type interne de l''objet	: 1 "
longueur du bloc	: 2 "
compteur de references	: 4 "
donn'ees	

Ces blocs ne contiennent pas de référence à d'autres objets.

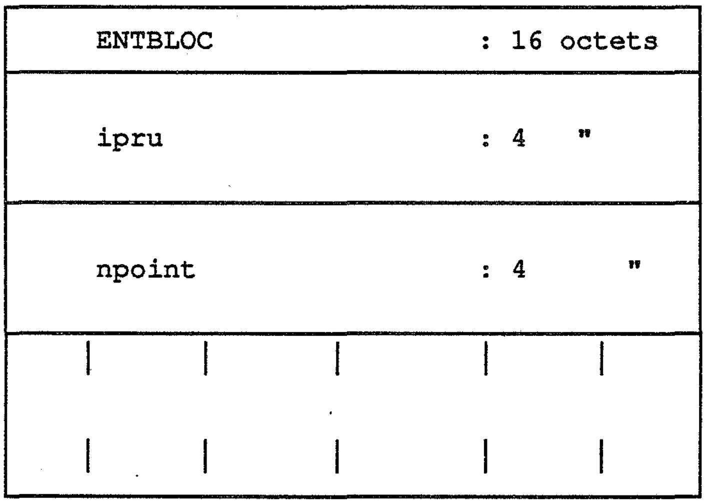
Les blocs de pointeurs contigus

Leur en-tête est analogue à celui des blocs sans pointeur, à deux informations complémentaires près :

ipru : indice de la première RU contenu dans l'objet .Cet indice indique le nombre de taille de RU par rapport au début de la zone utile de l'objet (c'est à dire juste apres l'entete).

npoint : nombre de pointeurs.

Ainsi le debut de ces blocs ont la structure suivante :



les pointeurs sont des réferences universelles. Ces blocs sont alloués à des objets de taille importante ou de type tableaux arborescents.

Le type interne du bloc permet de distinguer les 2 catégories de bloc :

GMBTDON : bloc sans RU
GMBTVRU : bloc contenant un "vecteur" de RU

2.3 ORGANISATION DE LA MEMOIRE SECONDAIRE

APL 90 utilise un (ou plusieurs) **espace virtuel** (dits espaces secondaires). Un espace est une zone disque contigue. Un espace peut avoir une taille arbitraire (>8k octets).

Si un espace occupe plus de 128Mo, il est partagé en **bancs** de 128 méga-octets. Le dernier banc d'un espace peut etre incomplet. Un banc est decoupé en 128 **quartiers** de 1 méga-octets. Un quartier est decoupé en 32768 **grains** de 32 octets. Le grain est l'unite d'allocation de la mémoire secondaire. Un **bloc** est un ensemble de grains contigus. Une **page** est un bloc de taille fixe. Dans la version 1, la taille d'une page est fixée a 128 grains, soit 4k octets. La page est l'unité de transfert disque-mémoire commune.

Dans chaque quartier, la **page zéro** sert de table d'allocation fine par grains pour le quartier. Il y a 256 pages par quartier et 128 grains par page, ce qui correspond à une table de 32768 bits. En outre, dans chaque banc, la **page un** sert de table d'allocation par pages pour l'ensemble du banc (128 quartiers de 256 pages).

2.4 STRATEGIE D'ALLOCATION DE LA MEMOIRE SECONDAIRE

Toute demande de taille mémoire arbitraire doit pouvoir être satisfaite. Les blocs de taille inférieure à une page (128 grains) sont alloués par nombre entier de grains. La technique d'allocation vise à ce niveau à optimiser l'allocation, d'une part en essayant de l'effectuer dans une (ou plusieurs) **page courante,** ce qui permet de rassembler des données logiquement liées entre elles, d'autre part en tentant d'allouer un bloc libre de taille égale à celle demandée, plutôt que de fragmenter le premier bloc rencontré de taille égale ou supérieure à celle nécessaire, ceci toujours dans la page courante.

Les blocs de taille supérieure à une page (4k octets) sont alloués par nombre entier de pages. Cette allocation se fait par recherche dans la table d'allocation du banc, ceci pour les blocs de taille raisonnable (par exemple, 8 ou 16 pages, soit 32 a 64 k octets). En cas de demande de bloc plus grand, la zone mémoire est fournie sous forme d'un **bloc paginé,** c'est à dire d'un bloc contenant une liste de pointeurs (réferences universelles) à d'autres blocs. Ces autres blocs sont tous de même taille, et correspondent à un nombre entier de pages.

3 GESTION DE LA MEMOIRE CENTRALE

3.1 TERMINOLOGIE

Pour pouvoir manipuler des objets APL, il est nécéssaire que ceux-ci soient disponibles en mémoire centrale (MC). Les petits objets sont transférés dans leur totalité depuis le disque, les gros objets sont accédés par morceaux, l'unité de transfert entre la mémoire secondaire (MS) et la MC étant la **page.** Seul un petit nombre de pages peut être disponible en MC. Un emplacement MC susceptible de recevoir une page est appelé **cadre.** C'est une zone mémoire centrale contigue dont les adresses de début et de fin sont fixes.

3.2 LA GESTION DES CADRES

La gestion des cadres nécessite la définition de certaines stratégies :

- Localisation et accès à un objet en memoire centrale.

- Allocation des cadres.

- Algorithmes de remplacement des pages en mémoire centrale.

Accéder à un élément d'un objet en mémoire MC, nécessite l'établissement d'une correspondance entre sa désignation (ou référence universelle) et une adresse de cadre contenant la page où réside cet élément.

3.2.1 TRADUCTION DYNAMIQUE DES ADRESSES

Il s'agit de trouver rapidement à partir d'une référence universelle, si une page est présente ou non en MC. Dans l'affirmative , accéder au cadre qui la contient.

Ceci nécessite l'implantation d'une **Table d'identification des pages** (TID) en MC. Rappellons que l'identité des pages est une adresse mémoire secondaire.

L'indice de chaque entrée de TID correspond au numéro de cadre où est logée la page espace secondaire dont l'adresse universelle est contenue dans l'entrée.

La recherche d'une page consiste à comparer séquentiellement l'identité de la page aux contenus des entrées de TID. Afin de minimiser les coûts de recherche nous avons implanter une fonction de hascodage qui utilise comme clé les certains bits de la référence universelle.

Nous avons défini une **table de hascodage** (THC), dont chaque entrée représente une classe de hascode et contient le numéro de cadre (entrée de TID) où se trouve le premier élément de la classe.

Afin de retrouver l'ensemble des éléments d'une classe, on utilisera une **table de collisions** (TDC). Cette table permet un chainage simple des entrées de TID de même hashcode.

Pour faciliter la mise en oeuvre de la pagination, une **table de contrôle des pages** (TCP) est necessaire. Elle contient des informations sur les entrées de TID : utilisation de l'entrée , modification de la page, verrous d'accès etc...

3.2.2 STRATEGIE DE REMPLACEMENT DES PAGES

Aux termes des analyses faites dans [Sako83], la stratégie de remplacement des pages en MC choisie est de type **lru.** C'est à dire qu'on remplace la page la moins recemment utilisée. Cet algorithme supporte convenablement le principe de localité et le modèle du " working - set ".

3.3 LES PRIMITIVES DE LA GESTION MEMOIRE

3.3.1 PRIMITIVES DE LA GDM

3.3.1.1 Primitive d'allocation **gmsalct(n,pru)**

n = nombres de grains mémoire a allouer

pru = pointeur sur la structure de type RU où sera rangée la RU attribuée a l'objet.

Cette primitive alloue de la mémoire brute dans l'espace dont l'identité est fournie par pru.

Stratégies d'allocation : L'allocation se fait par grains si n est inférieur à une taille définie dans la variable LAG. Sinon on alloue un nombre entier de pages immédiatement supérieur ou égal à la taille de l'objet.

En cas d'allocation fine (c'est à dire par grains) on essaiera de satisfaire la demande dans l'ordre suivant :

1- la page courante

2- le quartier courant

3- l'espace mémoire tout entier

Dans le deuxième cas on balaie la table d'allocation par grains (page zéro du quartier courant) en utilisant deux indices :

ipbit = indice du premier grains libre dans le quartier.

idbit = indice du dernier grains libre dans le quartier.

Dans le troisième cas ou en cas d'allocation par pages on balaie directement la page d'allocation par page (page UN de l'espace) et on alloue la ou les premières pages libres rencontrées.

Cette primitive éffectue un certain nombre de mise a jour :

ipbit,idbit,npct,nqct,tables d'allocation

Elle installe au début du bloc alloué une entête qui occupe 16 octets et qui est la concaténation des deux structures suivantes :

```
typedef struct {
        char typ1,typ2 ;
        short idev ;/*ident espace*/
        long adr ; /* adresse univ */
} RU ;
```

Ceci est la copie de la référence universelle

```
typedef struct {
        char semaph ;
        char typeint;
        short size ;
        long refcount;
} ATTRIB ;
```

La primitive gmsalct () initialise le compteur de référence de l'objet a 1 et copie la taille de l'objet (en nombre de grains) dans la zone attribuée a cet effet dans l'entete de bloc.Elle ne rend aucun résultat explicite.En cas de succès elle remplit le champ ADRS de la zone pointée par pru avec l'adresse universelle attribuée à l'objet.Sinon elle affiche un message d'erreur.

3.3.1.2 Primitive de désallocation gmslib(pru)

pru = pointeur sur la référence universelle

Cette primitive désalloue de la mémoire brute de façon primaire. Elle commence par matcher la référence fournie avec celle trouvée dans l'entete du bloc.

En cas de correspondance elle met un masque en début d'entete de l'objet et récupere sa place .

Elle ne rend aucun résultat explicite .En cas d'échec elle affiche un message d'erreur.

3.3.2 LES PRIMITIVES DE LA GDB

gmbcopy(prus,prud) :

copie physique d'un bloc source vers un bloc destinataire. Le compteur de references du bloc est initialise a 1. Si le bloc copie est de type 2 on applique sur toute sa descendance la primitive de duplication de bloc gmbdup(pru).

NOTA prus ou pru = pointeur sur a RU a fournir

prud = pointeur sur la RU resultant de l'action des primitives.

gmbdup(pru) :

demande de partage d'un bloc. C'est une copie logique.On incremente le compteur de references du bloc.

gmbmod(prus,prud) :
demande de modification d'un bloc .Si le compteur de references de l'objet vaut 1 on autorise la modification sur l'objet.Sinon il y a creation d'une copie physique sur laquelle aura lieu la modification.

résultat : 0 pas de copie physique
1 copie physique

gmbkill(pru) :
demande de destruction logique de l'objet.On decremente le compteur de reference.S'il devient nul alors on recupere la place physique de l'objet,en appliquant auparavant gmbkill(pru) sur la descendance de l'objet si celui est de type 2 .

gmbtype(pru,type) :
positionnement du type interne

gmbxtb(pru,nb,type) :
agrandissement d'un bloc de nb grains. Le bloc initial est recopié dans le nouveau puis détruit. (le paramétre type est inutilisé pour le moment.)

Les deux modules **gmbkill** et **gmbcopy** sont éventuellement récursifs si le bloc traité contient des RU. La récursivité est traitée en utilisant une table **tacru** donnant, pour chaque type de bloc, les adresses de deux fonctions:

- une fonction d'initialisation d'accès préparant une structure d'interface pour

- une fonction d'accès donnant l'adresse physique de la RU suivante, ou un pointeur nil en fin de bloc.

annexe 1

ANNEXE 2

Cette annexe, présente une grammaire pour un langage oriénté objet et compatible APL 2. Elle résulte des travaux de Jean Jacques Girardot et a été utiliser dans l'analyseur syntaxique du système APL 90.

```
    ∇ GRAMMAR
        ( ) ← → [ ]
        ⍝ GRAMMAR G20 : APL2-LIKE AND OBJAPL 90
        ⍝ JJG. 20/08/86 - VERSION 3
        ⍝ STATEMENT
        S    ::= A                        1
                 →   A                    3
                 →                        6
                 SE                      60
                 F                       61
                 FA                      62
        ⍝ STATEMENT SEQUENCE
        SE    ::= ◇                     63
                  ;                     64
                  ⍎                     65
                  ◇   A                 66
                  ;   A                 67
                  ⍎   A                 68
                  ◇   →  A              69
                  ;   →  A              70
                  ⍎   →  A              71
                  ◇   FA                72
                  ;   FA                73
                  ⍎   FA                74
                  ◇   F                 75
                  ;   F                 76
                  ⍎   F                 77
                  S   SE                78
        ⍝ FUNCTION SPECIFICATION
        FA    ::= ID  ←   F               9
        ⍝ ARRAY EXPRESSION
        A     ::= OS        1
                  F   A     4
                  OS  F   A     5
        ⍝ STRAND EXPRESSION
        OS    ::= B          1
                  OI        1
                  OS  B    18
                  OS  OI   18
```

```
[39]    ⍝ SIMPLE ARRAY EXPRESSION
[40]    OI     ::= FNI                   1
[41]               (   A   )             2
[42]               B   [   ]                  40
[43]               FNI [   ]                  40
[44]               (   A   )   [   ]          40
[45]               B   [   A   ]              41
[46]               FNI [   A   ]              41
[47]               (   A   )   [   A   ]      41
[48]               B   [   IL  ]              42
[49]               FNI [   IL  ]              42
[50]               (   A   )   [   IL  ]      42
[51]               ID  ←   A                  10
[52]               ID  [   ]   ←   A          43
[53]               ID  [   A   ]   ←   A      43
[54]               ID  [   IL  ]   ←   A      43
[55]               (   EA  )   ←   A          44
[56]               QQ  ←   A                  13
[57]    ⍝ NILADIC OBJECT
[58]    FNI    ::= FN                         16
[59]               QQ                         14
[60]    ⍝ SELECTIVE SPECIFICATION
[61]    EA     ::= SID                        45
[62]               F EA                       46
[63]               OS F  EA                   47
[64]    ⍝ VECTORIAL (STAND) SPECIFICATION
[65]    SID   ::= ID                          48
[66]              SID  ID                     49
[67]    ⍝ FUNCTION EXPRESSION
[68]    F      ::= P                           1
[69]               FI                          1
[70]               F   GP1                    28
[71]               OS  GP1                    23
[72]               F   GP2    P               50
[73]               F   GP2    FI              50
[74]               F   GP2    B               51
[75]               F   GP2    OI              51
[76]               OS  GP2    P               52
[77]               OS  GP2    FI              52
[78]               OS  GP2    B               53
[79]               OS  GP2    OI              53
[80]    ⍝  SIMPLE FUNCTION EXPRESSION
[81]    FI     ::= ∘                           1
[82]               MSG                         1
[83]               (   F   )                   2
[84]               (   FA  )                   2
[85]               P   [   A   ]              25
[86]               (   F   )   [   A   ]      25
[87]               (   FA  )   [   A   ]      25
[88]    ⍝  MONADIC OPERATOR
[89]    GP1    ::= OP1                         1
[90]               (    GP1  )                  2
[91]               OP1    [    A    ]          32
[92]               (    GP1  )   [   A    ]    32
```

```
[93]     ⍝  DYADIC OPERATOR
[94]     GP2    ::= OP2                            1
[95]                (    GP2   )                   2
[96]                OP2    [   A     ]             31
[97]                (    GP2   )     [     A     ] 31
[98]     ⍝  INDEX LIST
[99]     IL     ::= ;                             22
[100]               ;    A                        22
[101]               A    ;                        22
[102]               A    ;    A                   22
[103]               IL   ;                        22
[104]               IL   ;    A                   22
    ∇
```

Les symboles terminaux utilisés dans cette grammaire sont :

- *B*: est un tableau de base (une constante ou une valeur d'une variable).
- *FN*: fonction niladique ou ⎕ ou ⍞.
- *P*: fonction primitive ou définie.
- *QQ*: est le ⎕ ou le ⍞ placé à gauche de ←
- Les autres synboles représentent leurs propres valeurs sauf { et } qui symbolisent les crochets d'axe.

  Les symboles non terminaux sont :

- *S*: une intruction APL valide.
- *A*: une expression de tableau.
- *OI*: le paramètre gauche d'une fonction.
- *IL*: une liste d'indices non vide.

Chaque ligne décrit une règle unique. Le symbole `::=` sert à separer les parties droites des parties gauches des règles. Lorsque ce symbole est absent, la partie gauche d'une règle est identique à celle de la précédente. La première ligne contient l'ensemble des symboles auquels aucune valeur n'est associée à l'exécution. Les nombres qui apparaissent à la fin de chaque ligne indique l'action sémantique associée à la règle.

Les lignes 12 et 27 introduisent différents séparateurs d'instructions tels que : ; ou ⋄. La *stand notation* est décrite de la 35 à la ligne 38. Outre la spécification de fonction, la grammaire offre la transmission de message (ligne82) utilisée dans OBJAPL 90.

28/1/1987

annexe 2

RESUME

Malgré l'évolution des techniques et des environnements de programmation, les systèmes APL existants souffrent encore de certaines limitations. L'approche d'APL 90 a consisté à mettre en évidence ces faiblesses et à proposer certaines solutions. Ce qui nous a amené à définir une architecture logicielle de machine que nous appelons ALM 90, architecture autour de laquelle fut bâti le système APL 90.

Au fur et à mesure que les systèmes APL évoluaient, plusieurs propositions d'extensions sont apparues. Les extensions qui nous semblent essentielles (tableaux généralisés, "strand notation" et opérateurs) ont été analysées en vue de leur intégration à notre système.

La programmation orientée objet suscite de plus en plus d'intérêt de nos jours. Beaucoup de langages procéduraux proposent actuellement des extensions orientées vers la programmation objet. Nous nous sommes intéressés aux concepts de ce paradigme. Ce qui nous a conduit à définir un modèle orienté objet pour le langage APL que nous appelons OBJAPL 90. Ce modèle a été élaboré avec une vision des objets analogue à celle de SMALLTALK.

APL 90 est un système qui propose à l'utilisateur de nouveaux types primitifs (atomes), de nouvelles structures de données (tables hash-codées), et de nouvelles méthodes de programmation expérimentales, grâce à son extension objet.

Ecrit en C et opérationnel sous UNIX, APL 90 est disponible sur de nombreux ordinateurs (SM 90, SPS 7, Macintosh, HP 9000 ...)

Mots clés :

APL, référence universelle, tableau généralisé, "strand notation", transmission de messages, SMALLTALK, langage orienté objet, programmation orientée objet